A formal foundation for the Dafny verifier

Benjamin Bonneau
Internship at ETHZ, February-July 2021
supervised by Gaurav Parthasarathy and Thibault Dardinier

Introduction

Dafny is a verification language [5]. An existing implementation allows to verify the correctness of a program and to compile it to several back-end languages. The verification is done by emitting a translation of the program in the Boogie intermediate verification language [4]. The Dafny verifier then accepts the original Dafny program if the Boogie verifier accepts its translation.
The semantics of a fragment of the Boogie language has been mechanised in Isabelle [3] during a work done in the programming methodology group of ETH Zurich [8]. This work also includes a proof generation tool that can certify the correctness of an execution of the Boogie verifier. Hence if one could prove the correctness of the translation from Dafny to Boogie, one could certify the correctness of Dafny programs.
During my internship, I formalised the semantics of a fragment of Dafny in Isabelle (section 1) and proved several results that would help proving the correctness of the translations (section 2).

1 Language and semantics

1.1 Language

We only support a fragment of the Dafny language. It is defined in `DafnyLang`.

```
datatype prim_ty = TInt | TBool
datatype ref_ty  = TClass class_ident | TArray
datatype ty      = Tprim prim_ty | TInd indty_ident | Tref ref_ty
datatype type    = TCon ty "type list" | TVar nat

record ('t, 'a) ty_inst =
  BaseType :: 't
  TypeArgs :: "'a list"
type_synonym ref_type = "(ref_ty, type) ty_inst"
```

We support the boolean and integer primitive types and the class types. Our definitons also mention the array types and the inductive datatypes but they are not implemented in the semantics. I tried however to take the inductive datatypes into account in some definitions. A `type` is either a type variable which refers to a type present in the context or a type constructor `ty` applied to a list of type parameters. The primitives types take no type parameters and the class and inductive types accept any list of parameters whose length is the number of type parameters declared for the type. Contrary to the complete language, our fragment does not put any restrictions on the type parameters, we assume that any type is equality-supporting and auto-initializable (Dafny Reference Manual[6], section 8). When we need to constrain the kind of `ty` used, we use a `ty_inst` record instead of `TCon`. For instance `ref_type` represents a reference type, that is, a class type or an array type.
The declarations of classes and inductive datatypes are stored in the `declarations` record:

- `class_ntyps :: "declarations ⇒ class_ident ⇀ nat"` gives the number of type parameters of a class and `class_fields :: "declarations ⇒ class_ident ⇒ cfield_ident ⇀ type"` its declared fields and their types. In our fragment, classes do not have any method or constructor and there is no inheritance mechanism. In fact, those two functions uses the same field of the `declarations` record (`Class`) but in practice it is more convenient to use those wrappers.

- `IndTy :: "declarations ⇒ indty_ident ⇀ nat"` gives the number of type parameters of an inductive datatype and `IndCt :: "declarations ⇒ constr_ident ⇀ (indty_ident × type list)` the type returned by a datatype constructor and the types of its parameters.

The expressions are defined as follows:

```
datatype literal  = LitInt int | LitBool bool | LitNull

datatype unary_op = Not | Neg | Allocated

datatype bin_op =
    Eq   | Neq
  | Add  | Sub  | Mul | Div
  | Le   | Lt   | Ge  | Gt
  | Or   | And

datatype expr =
    Literal  literal
  | Var      var_ident
  | UnaryOp  unary_op expr
  | BinaryOp bin_op   expr expr
  | MembrSel expr     cfield_ident
  | Old      expr
```

Our fragment includes the literals, the variables, a subset of the unary and binary operators, the class field selections and the `old` operator. Our definition includes the `Neg` unary operator but in the `C#` implementation of Dafny, it is replaced by a subtraction and is not represented in the Abstract Syntax Tree (AST). We include the `allocated` unary operator (which evaluate to true if the inner expression reduces to a value that contains only allocated references, null is considered allocated here) because its presence has an impact on some proofs. Our subset of binary operators include equality testings and some operations on integers and booleans. In particular, it contains a partially-defined operation (the division) and some lazy operators. The division is the euclidean one.

```
type_synonym frame = "expr set"
datatype wild_frame = WildFrame | Frame frame
```

A frame denotes a subset of the heap. In Dafny, frames are used to restrict the locations that a method, function or statement is allowed to read or write. Since our fragment does not include collections types, a frame can only be a set (finite in practice) of expressions that should reduce to references. When it is allowed, a `WildFrame` represents the absence of a framing condition. In the case of a write frame, the impacted statement is always implicitly allowed to modify any location that is allocated during its execution (even if it is not mentioned in the modifies clause). The complete Dafny language allows the use of expressions reducing to a collection of references. It also allows frames to restrict modifications to certain fields of references (instead of all fields) with a special syntax.

The statements are defined as follows:

```
record LoopSpec =
  LoopInv  :: "expr list"
  Mod      :: wild_frame

datatype stmt =
    Block   "stmt list"
  | VarDecl "var_ident ⇀ type"
  | Update  "expr list" "expr list"
  | New     expr "(class_ident, type) ty_inst"
  | Assert  expr
  | Assume  expr
  | If      expr stmt stmt
  | While   expr stmt LoopSpec
  | Break
  | Call    "expr list" method_ident "type list" "expr list"
```

The local variable declarations are represented as a `VarDecl` statement. We do not include them in the block definition because the translation produces some commands (some `havoc`) at the location of the declaration. When defining a Dafny program, the declarations are represented as a list of pair of `var_ident` and `type` but the semantics only uses the mapping obtained from this list (with `map_of`). The `Update`, `New` and `Call` statements uses some left-hand sides represented as expressions. The head of an expression used as a left-hand side must be `Var` or `MembrSel`. In the AST of the `C#` implementation, the updates and allocations are represented with the same classes: `UpdateStmt` and `AssignStmt` in the case of a single assignment. In particular, it is possible to include allocations in a multi-assignment (if no constructor is called). In this fragment, I separated the `New` statement and only allowed single allocations because the semantics implemented by the Dafny verifier for multiple allocations is not completely invariant by permutation of the left-hand sides and right-hand sides. Since the two statements are still similar, it could have been better to keep a single statement and only consider allocations in single updates when needed. This could factorise some parts of the classical big-step semantics but this factorisation is already done in the pretty big-step semantics (section 2.1). The loops can be annotated with some invariants and a write frame. If such frame is not specified (with the keyword `modifies`) the loop has the same write frame as the enclosing context. This is represented with a `WildFrame` in our AST. In fact, the write frame could be removed from the loops and introduced with a `modify` statement. This would result in the same semantics and it seems that it would not be a problem for the translation.

The methods are defined with the following record:

```
record df_method =
  TypeParams :: nat
  Ins  :: "type var_decl list"
  Outs :: "type var_decl list"
  Req  :: "expr list"
  Ens  :: "expr list"
  Mod  :: "frame"
  Impl :: "stmt option"
```

The `declarations` record contains a map (`Method`) from method identifiers to method declarations. A method declaration includes:

- Its type signature, that is, its number of type parameters `TypeParams` and the types of its arguments (`Ins`) and return values (`Outs`). Each argument and return value has an identifier that can be used to refer to it in the specification.
- An ordered list of preconditions (`Req`) declared with the keyword `requires`. Each precondition is a boolean expression that may depend on the values of the arguments and the sate of the heap when the method is called. In a correct program, the caller must ensure that each of them is satisfied.
- An ordered list of postconditions (`Ens`) declared with the keyword `ensures`. Each postcondition is a boolean expression that may depend on the argument and return values and the state of the heap at the beginning of the call and at its end. When called on correct arguments (that is, satisfying the preconditions), the method should satisfy them when it returns (if it returns) and the caller can assume it.

- A write frame (`Mod`) declared with the keyword `modifies`. It defines a subset of the heap that the method is allowed to modify. In a correct program the caller must ensure that it is itself allowed to modify this subset of the heap. When the method returns, the caller can assume that the method has not modified any part of the heap that it was not allowed to. Like the preconditions, the modifies clause can depend on the value of the arguments and the state of the heap at the beginning of the method.
- Finally, an implementation can be provided (`Impl`). It should satisfies the specifications of the method. The main task of the Dafny verifier is to check it.

I implemented a module for Dafny that exports the Isabelle representation of a program (only a subset of the language is supported) [1].

1.2 Semantics

The values used in the semantics are defined in `DafnyValues`:

```
datatype val = VInt int | VBool bool
             | VIndCons constr_ident "itype list" "val list" | VRef ref
```

The `VIndCons` constructor is used to build datatype values. The two first arguments determine which constructor has been used to build the value and the third argument is the list of parameters given to this constructor. The `VRef` constructor is used for the values of reference types, that is, classes and arrays. The `ref` type could be an arbitrary type with an infinite number of inhabitants and a particular `NULL` value. In Isabelle, we fixed this type to a datatype isomorphic to `nat`. Values have closed types given by `val_typ :: "declarations ⇒ (ref ⇀ iref_type) ⇒ val ⇒ itype ⇒ bool"`. The `itype` type (instantiated type) is used to represent closed type. Using this different type allows to remove some hypothesises and also avoids confusing the types of the values with the types used in the language. It is defined similarly to `type` but without type variables:

```
datatype itype = ITCon ty "itype list"
```

The Dafny semantics uses a heap to store the values of the reference types. We represents it as a map from references to heap locations:

```
datatype ref_field = CField cfield_ident | AIndex nat
type_synonym heap_loc = "(iref_type × (ref_field ⇀ val)) option"
type_synonym     heap = "ref ⇒ heap_loc"
```

A heap location is either `None` if it is not allocated or `Some (rty, fs)` where `rty` is the type of the object stored at this location and `fs` contains the values stored in its fields.

The frame values are represented with `frame_val = "(ref × ref_field) set"`. In our fragment all the frames values have the shape `O × UNIV` since we only provide a syntax with object granularity. To ensure that the write frames include the locations allocated during the execution of the statements, we add to the frames any location that is not allocated when we evaluate the value of the frame.
A left-hand side value is either a variable or a field of a heap location:

```
datatype lhs_val = LhsVar var_ident | LhsField ref cfield_ident
```

A normal state of a Dafny program is represented with the `nstate` record:

```
type_synonym var_state = "var_ident ⇀ val"

record 'h nstate_h =
  VarState :: var_state
  Heap     :: 'h
  OldHeap  :: 'h
type_synonym nstate = "heap nstate_h"
```

The `nstate_h` record is defined for any type of heaps in order to be reused in the proofs of the translation. The front-end semantics only uses `nstate`. A normal state consists of a `VarState` that map variables to their values and two heaps, the one at the beginning of the method and the current one. A general state of a program is then either:

- A normal state `NState (n :: nstate)`
- A break state `BreakState (n :: nstate)`, which is used to implement the `Break` semantics. This state is propagated upward in the AST until we find an enclosing loop where we return to a normal state.
- An exception state `ExcState (e :: exc_kind)` which is either a `FailState` or a magic state. The first one is used to represent an error during the execution (for instance an assertion whose guard is not satisfied), the second one is emitted if the execution reaches a state that is assumed to never happen (for instance an assumption whose guard is not satisfied). For the correctness, we simply ignore the magic states, their interest would be to ensures that a well-typed program always reduces (it is not the case with the current semantics). In any case, an exception state interrupts the execution by being propagated to the root of the AST.

Our front-end semantics of Dafny is defined in `DafnySemantic`. Our reductions depend on a context. This context includes the type parameters passed to the method as a list of instantiated types (`TypeParams`). Those type parameters are used to instantiate the types that appear in the AST. When considering the reduction of an expression we use an `expr_context` which also contains a `Reads` frame value. For the statements we use `stmt_context` which contains a `Modifies` frame value. Since statements are always allowed to read the entire heap, we obtain an expression context from a statement context by choosing `UNIV` as a `Reads` frame. In the complete Dafny language, the expressions used to define the functions could be reduced with a `Reads` frame different from `UNIV`, but it is never the case in our fragment.
The reduction of expressions is defined by the predicate `expr_red :: "declarations ⇒ expr_context ⇒ expr ⇒ nstate ⇒ val option ⇒ bool"`. It takes as parameters the declarations of the program and the context and state where the expression is evaluated. An expression reduces to `Some v` if it is defined and to `None` if it is undefined. Evaluating an undefined expression is an error which is propagated by `expr_red` and leads to a `FailState` in our semantics for the statements. If we encounter a typing error, we do not reduce the expression at all, hence it is possible for ill-typed expressions to be neither defined nor undefined.
We simply reduce literals to the corresponding values. A variable is reduced to the corresponding value of the `VarState`. The absence of a variable is considered as a typing error and produces no reduction. The reduction of the unary operators is standard, for the `Allocated` operator, we consider `NULL` as an allocated value. The binary operators are more interesting. For the strict operators, we reduce both sides (and propagate eventual errors) then:

- if the operation is not well-typed, we do not reduce.
- if the operation is defined for the values obtained, we reduce to its result.
- if the operation is not defined, we reduce to `None`. In our fragment, this can only happen for the division operation.

For the lazy operators, we start by reducing the left expression (and propagate an eventual error). Then, if the operator has a lazy behaviour for this value, we immediately reduces to the value given by this laziness (for instance if the left expression of an `Or` reduce to `VBool True`, the binary operation reduces to `VBool True`). Otherwise we reduces the right expression and continue as for the strict operators. The

laziness is important because it allows the operator to reduce even if the second operation is undefined. For instance `x > 0 && 100/x >= 50` is always defined.
For the field selection operator, we expect the inner expression (the object) to reduce to a `VRef r`. Then we check that the corresponding location is allocated in `Heap` and that it belongs to the to the `Reads` frame (for the field considered). If one of those two conditions is not satisfied we reduce to `None`. Otherwise we return the value of the field in the heap location (the absence of this field is considered as a typing error).
For the `Old` operator, we reduce the inner expression in the state where we replace the current heap by the old heap and propagate its result.
The expression reduction is deterministic, that is, for some fixed declarations, context and state, an expression reduces to at most one `val option`.

Using the reduction of expressions, we can define the reductions of frames defined using a set of expressions. A such frame is not defined if at least one expression is not. Otherwise we first compute the union of the locations `{r}` × `UNIV` such that an expression reduces to `Some (VRef r)`. We then add to the frame any location that is not allocated in order to allow affected statements to modifies the objects that they allocate.
For the left hand sides a `Var x` always reduces to a `LhsVar x` without premises. The reduction of a field selection `MembrSel obj f` is similar to the right-hand side case (when the value of the field is red). We expect the inner expression to reduce to a reference `VRef r` and we fail if the corresponding location is not allocated or does not belong to the `Modifies` frame, otherwise the left-hand side reduces to `LhsField r f`.

The reduction of the statements is defined with a big-step semantics by `stmt_red :: "declarations ⇒ stmt_context ⇒ stmt ⇒ nstate ⇒ state ⇒ bool"`. This predicate expresses the fact that, starting from the normal state at the beginning of the statement, the program can exit the statement in the second state.
To reduce a non-empty block, we reduce the first statement. If the resulting state is a `BreakState`, a `FailState` or a magic state, the whole block reduces directly to this state. Otherwise it is a normal state and we use this state to reduce the remaining statements recursively. An empty block has not effect. Note that we do not do any modification to the state when we exit the block. In particular we do not remove the variables declared inside the block. The typing ensures that any variables declared in the same branch of the program have distinct identifiers, hence there is no shadowing.
For a variable declaration `VarDecl ts`, we assign to the affected variables any arbitrary values of the correct types. The `Assert` and `Assume` statements reduces their guards (and reduces to `FailState` if the guard is not defined). They have no effect if the guard reduces to true. If the guard reduces to false, an assertion reduces to `FailState` and an assumption to the magic state.
For an `Update` we reduces the left-hand side and right-hand side in the entry state. If this statement tries to assign two different values to a same left-hand side value, it fails. Otherwise we assign the right-hand side values to the left-hand sides. A `New` statement firstly reduces its left-hand side. It then allocates an arbitrary unallocated location of the heap. The type given to this location is the class type argument of the `New` statement instantiated with the type parameters of the context. The semantics ensures that any field declared for this class is initialised in the newly allocated locations with a value of the correct type in the state *after* the allocation (remember that `val_typ` depends on the types on the heap) according to Dafny Reference Manual[6], 13.3.2.1 and the behaviour of the verifier. The left-hand side is finally assigned with a reference to this location.
An `If` statement reduces like its `then` branch if its guard reduces to `VBool True` in the initial state and like its `else` branch if its guard reduces to `VBool False`.
For a `While`, we first compute the value of the associated write frame and check that it is a subset of the context `Modifies` frame. We then reduce the loop with this new write frame. We start by checking that the invariant holds. Then, if the guard reduces to `VBool False` the loop has no effect. If it reduces to `VBool True`, we reduces its body. We propagate the `FailState` and the magic state. If it reduces to a `BreakState s`, the loop reduces to `NState s`. Finally if it reduces to a normal state, we repeat starting from the invariant check. Since the reduction is defined as an inductive predicate, we only consider finite derivation trees so we do not take infinite loops into account. A `Break` statement reduces to a `BreakState` which is then propagated to the first enclosing loop and makes it stop.

In our fragment, the correctness of a program consists of two things:

- The specifications of any declared method are well-formed, that is, the expressions involved never produce an error.
- The implementations provided satisfy the specifications and never encounter errors.

In this work, I only focused on the verification of the second part. The first part is assumed and used to remove some well-formedness checks.

2 Proving the correctness of the translation

Given a translation made by the Dafny verifier, that is, a Dafny program and its Boogie translation, we want to prove that the correctness of the Boogie program implies the correctness of the Dafny one. Since there is not yet a mechanised semantics for Boogie ASTs, I decided to consider directly the translation from Dafny to Boogie CFG.
In order to check the correctness of methods, Dafny emits three Boogie procedures for each method `m`:

- `CheckWellformed$$m` is used to check that the method specifications are well-formed.
- `Call$$m` is used to translate the calls to this method. It is independent of the optional method implementation and has the most general behaviour satisfying the method specifications.
- `Impl$$m` is used to check the correctness of a provided implementation.

Since I did not implement the translation of calls and I assume the well-formedness of specifications, the only procedure that I consider is `Impl$$m`. We expect that the correctness of this Boogie procedure, which we assume, implies the correctness of the Dafny implementation of the method. By contraposition, we can obtain this correctness by proving that, if there was a failing trace in the Dafny implementation, there would be a failing trace for the Boogie procedure. Hence the goal of our proof strategy is, given a failing trace for the Dafny implementation, to obtain a failing trace for the Boogie procedure.
The adequate tools to achieve this are simulations [7]. The idea is to define a simulation relation between the states of the source semantics and the states of the target semantics. By proving that we are able to simulate individual steps of the source semantics by some steps in the target semantics, we can obtain a trace in the target semantics such that each state of the original trace is related to a corresponding state. In particular the end states of the two traces are related, hence if the relation preserve failing states, we will have found a failing trace in the target semantics.
Instead of attempting to do this proof directly, I defined two others semantics for Dafny statements (the pretty semantics and the intermediate semantics) and I translate successively the original trace in the different semantics. Hence there are three steps, each of them follows a simulation strategy. The first two steps are simulations between semantics defined on the same Dafny implementation and are proven once and forall. The goal of those additional steps is to do the final simulation to Boogie from a semantics which is closer to the behaviour of the translation.

2.1 First step: the pretty semantics

The goal of the first step is to obtain a reduction in a semantics whose structure is closer to the structure of the translation. For instance, the font-end semantics do not fix an order for the reductions of the left-hand sides and right-hand sides of an `Update` (figure 1) but in the translation in Boogie, the left-hand sides are reduced before the right-hand sides. The consequence is that a rule of the Dafny semantics allows to reduce directly to `Failure` if an error is encountered in the right-hand sides. However in the Boogie translation, one must first execute the left-hand side part before evaluating the right-hand sides. Hence, to simulate this case, one must prove that the commands emitted for the left-hand side part reduces (possibly to an error). One can either use the Boogie type safety or the Dafny one.
In this first proof step, we use the Dafny type safety to discharge some of those transformations. We do not change anything for the expressions, their semantics is the same at both ends. For the statement the target semantics takes more hypothesis for a same reduction. I have chosen to define this target semantics (in `DafnyPrettySemantic`) as a pretty big-step semantics [2]. The main idea of a pretty big-step semantics is to use atomic rules in order to factorise some parts of the rules of a same statement and even to define rules that apply to different statements. To achieve this, one needs to introduce more statements to represents the states between the successive atomic reductions. Hence our pretty semantics is defined

$$\frac{\texttt{lhss} \Downarrow \text{fail} \ \lor \ \texttt{rhss} \Downarrow \text{fail}}{\texttt{Update lhss rhss} \rightarrow \texttt{Failure}}\,\texttt{PFailUpdt} \qquad \frac{\texttt{lhss} \Downarrow \texttt{lhss\_v} \quad \texttt{rhss} \Downarrow \texttt{rhss\_v} \quad \texttt{update(lhss\_v,rhss\_v)}\ \text{fails}}{\texttt{Update lhss rhss} \rightarrow \texttt{Failure}}\,\texttt{FailUpdt}$$

$$\frac{\texttt{lhss} \Downarrow \texttt{lhss\_v} \quad \texttt{rhss} \Downarrow \texttt{rhss\_v} \quad \texttt{update(lhss\_v,rhss\_v)} \rightarrow \texttt{s'}}{\texttt{Update lhss rhss} \rightarrow \texttt{NState s'}}\,\texttt{RedUpdt}$$

Figure 1: Informal reduction rules for the `Update` statement in the front-end semantics.

on another statement type `ystmt`. There is an injection `stmt_to_pretty` from the front-end statements to `ystmt`. The `ystmt` for the update are the following:

```
datatype ystmt =
  | YUpdt0 "expr list" "expr list"
  | YUpdt1 "lhs_val list" "expr list"
  | YUpdt2 "(lhs_val × val) list"
  ...
```

An `Update` is mapped to an `YUpdt0`, the others `ystmt` store the already reduced values so that one can handle the reduction of a subpart with only one set of rules and use the result in multiple rules. The reductions rules for an `YUpdt0` can be visualized with the figure 2. Each arrow of this diagram represents a rule of the pretty semantics. For instance, the following informal rule reduces the right-hand sides:

$$\frac{\texttt{rhss} \Downarrow \texttt{rhss\_v} \quad \texttt{YUpdt2 (zip lhss\_v rhss\_v)} \rightarrow \texttt{s'}}{\texttt{YUpdt1 lhss\_v rhss} \rightarrow \texttt{s'}}\,\texttt{YRedUpdtRhs}$$

The reductions of the left-hand sides and right-hand sides that were occurring twice in the front-end semantics (in `FailUpdt` and `RedUpdt`) appear only once in the pretty semantics. Moreover, I was able to factorize the reductions of most of the expressions and left-hand sides of all statements using only four rules by defining functions that return the sub-expression (or left-hand sides) that has to be reduced in a `ystmt` (if any) and the `ystmt` that continues the reduction. As we wanted, the pretty semantics reduces the left-hand sides before the right-hand sides. In the translation to Boogie, we get the extra assumption that the left-hand sides reduces normally when we handle the case of an error in the reduction of the right-hand sides. Hence we only have to prove that there is a reduction to `Failure` from the node *after* the commands that handle the left-hand side reduction since we prove that this hypothesis implies the reduction of those commands.

$$\begin{tikzcd}
\texttt{YUpdt0} \arrow[rr, "\texttt{lhss} \Downarrow \texttt{lhss\_v}"] \arrow[dr, "\texttt{lhss} \Downarrow \text{fail}"] & & \texttt{YUpdt1} \arrow[rr, "\texttt{rhss} \Downarrow \texttt{rhss\_v}"] \arrow[dr, "\texttt{rhss} \Downarrow \text{fail}"] & & \texttt{YUpdt2} \arrow[rr, "\texttt{update(lhss\_v,rhss\_v)}\ \text{succeeds}"] \arrow[dr, "\texttt{update(lhss\_v,rhss\_v)}\ \text{fails}"] & & \text{success} \\
& \text{failure} & & \text{failure} & & \text{failure} &
\end{tikzcd}$$

Figure 2: Representation of the reduction rules for an `YUpdt0`

Another consequence of the use of a pretty semantics is that one need to define a meaningful hypothesis for each `ystmt` when reasoning by induction on a reduction in the pretty semantics. Hence, when proving the correctness of the translation of the `While` statement, I needed to define an induction hypothesis for each `ystmt` used during the loop. In fact, this simplified the proof compared to a previous attempt to do the translation directly from the front-end semantics. The reason is that each `ystmt` can be associated to a node in the Boogie program, so the hypothesis associated to the `ystmt` is simply the existence of a reduction in Boogie starting at this node. For each case of the proof, one only has to build the part of the reduction from the node associated to the current `ystmt` to the node associated to the next `ystmt`. With the front-end semantics one has only one induction hypothesis but each proof case requires to build the complete reduction from the entry of the loop to the next loop iteration, the exit or `Failure`. In practice I had defined some lemmas to handle the parts that were duplicated in the front-end semantics and most of those lemmas were used only once in the pretty semantics so I could inline them.
This need of an induction hypothesis for each `ystmt` was an issue when proving the type preservation of the pretty semantics. Indeed some statements contain some reduced values. But the typing judgement for Dafny values depends on the state of the heap so one cannot simply use a static typing judgment independent of the state for all `ystmt`. The solution that I have found is to add additional typing hypothesis in the premises of the rules of the pretty semantics. For instance the reduction of an `YUpdt2`

requires the corresponding left-hand sides and right-hand sides to have the same type. Those additional hypothesis are enough to ensure the type preservation of the pretty semantics without any hypothesis on the statements other than their reduction.
The simulation between the front-end and pretty semantics is a refinement [7], that is, the simulation relation is a function (the identity in this case) from states of the front-end semantics to states of the pretty semantics. This proof is done in `DafnyTypeSafety` and uses the type preservation of the front-end semantics to prove the reduction of some expressions (for instance the left-hand sides of `Update`) to change the structure of the reduction and to prove the additional hypothesis of the pretty semantics. This type preservation is deduced from the type safety of the pretty semantics and the existence of a simulated reduction (obtained as an induction hypothesis).

2.2 Second step: the intermediate semantics

The goal of the second step is to simulate the reduction with a partial heap (like in the front-end semantics) by a reduction with a complete heap (like the encoding in Boogie). The difficulty is the allocation statement. Indeed, for this statement the front-end semantics changes the type and fields of the allocated location whereas in Boogie the fields are not changed and the type is assumed (figure 3). Hence the locations that are allocated during the execution need to be initialised with the correct type and fields at the beginning of the Boogie trace.
In order to simplify the proof of the final simulation and since some of the functions defined in Boogie (in particular `dtype`) depend of the types of the heap locations, I have chosen to handle this change as a separate step. I have defined the intermediate semantics (`DafnyIntermediateSemantic`), which has the same rules as the pretty semantics except that it uses a complete heap. In particular, the intermediate semantics has a rule for the allocation that reflects the translation in Boogie. The type of heap locations in the intermediate semantics is:

```
record heap_loc =
  hl_state  :: "(bool × iref_type) option"
  hl_fields :: "ref_field ⇒ val"
  hl_fldprs :: "ref_field ⇒ bool"
abbreviation hl_alloc :: "heap_loc ⇒ bool" where
  "hl_alloc l ≡ ex_Some (hl_state l) fst"
abbreviation hl_type  :: "heap_loc ⇀ iref_type" where
  "hl_type  l ≡ map_option snd (hl_state l)"
```

A heap location has a boolean marker indicating if it is allocated, a type and some fields values. In practice a location can have `None` as `hl_state` (allocation and type). This is used to avoid needing to initialise the locations that are never used during the execution. A heap in the intermediate semantics is a complete mapping from references to `heap_loc`. We can define a relation between the partial heaps of the front-end semantics and those complete heaps. A partial and a complete heap are related (with `tr_heap`) if each of their corresponding locations `lp` and `lc` satisfies the following conditions:

- They have the same allocation status. That is `lp` is `None` if and only if `hl_alloc lc` is false.
- If they are allocated, they have the sames types and their fields satisfies the following conditions:
  - `hl_fields lc` extends the fields `flds` of `lp` (which is a partial mapping). That is, if a field is present in `flds`, it has the same value in `hl_fields lc`.
  - `hl_fldprs lc f` (field present) is true if and only if `f` is present in `flds`.

The purpose of `hl_fldprs` is to ensure that each complete heap is related to exactly one partial heap (otherwise it would be related to multiple partial heaps). Thanks to this one can define define a function `heap_of_itm` which inverts `tr_heap`:

$$\texttt{tr\_heap hp hc} \iff \texttt{hp = heap\_of\_itm hc}$$

This is convenient for the proof of the simulation because, by extending this function to states, one can define the reduction of expressions in a state of the intermediate semantics as the reduction in the state of the front-end semantics obtained with this function.
In order to know at the beginning of the trace which types and fields values to give to the locations of the heap that will be allocated, we use a backward translation. In fact the step is a hybrid backward-forward

translation because one needs to ensure that at the beginning of the trace built in the intermediate semantics, the old heap is equal to the heap. Given a reduction of a statement `stmt` in the pretty semantics between two normal states (vs_0, h_0, oh) (a normal state with variable state vs_0, heap h_0 and old heap oh) and (vs_1, h_1, oh) I prove the following result:

$$\begin{aligned}&\forall h_1'.\ \texttt{tr\_heap}\ h_1\ h_1' \Rightarrow \exists h_0'.\ \texttt{tr\_heap}\ h_0\ h_0'\\ &\qquad \wedge\ \forall oh'.\ \texttt{tr\_heap}\ oh\ oh' \Rightarrow (vs_0, h_0', oh')\text{ reduces to }(vs_1, h_1', oh')\end{aligned}$$

Where the obtained reduction is done on `stmt` in the intermediate semantics. In practice we also handle the reductions to the other kinds of states and we maintain some invariants on the heaps so that their translations to Boogie satisfies some typing requirement and axioms. In the formalism of [7] a hybrid backward-forward simulation uses a relation between states of the source semantics (the pretty semantics here) and set of states of the target semantics (the intermediate semantics here), that is a predicate $g : S \to \mathcal{P}(S') \to \text{bool}$ where S (resp. S') are the set of states of the automat associated to the program with the source (resp. target) semantics (those states should also include the location in the AST). Here, a state s (vs, h, oh) would be related with set of states (at the same location in the AST):

$$\big\{(vs, h', oh') \mid oh'.\ \texttt{tr\_heap}\ oh\ oh'\big\} \qquad \text{where } \texttt{tr\_heap}\ h\ h'$$

The result that has to be proven is the following for a reduction $s_0 \xrightarrow{s} s_1$ in the source semantics:

$$\forall G_1.\ g(s_1, G_1) \Rightarrow \exists G_0.\ g(s_0, G_0)\ \wedge\ \forall s_0' \in G_0.\ \exists s_1' \in G_1.\ s_0' \xrightarrow{t*} s_1'$$

The result that we prove implies this one. We have removed the last existential quantifier by choosing the same old heap (that is $s_1' \in G_1$) as in s_0 since we know that the reductions preserve the old heap. One may need this quantification if we added the `old@` (labelled old) operator.
In practice, to build the reductions in the intermediate semantics I apply the following strategy for each atomic step. Given a reduction between two normal states s_0 and s_1 and a state s_1' related to s_1 (that is, a state of the intermediate semantics that has the same variable state as s_1 and heap and old heap related to the heap and old heap of s_1), I consider the state s_0' related to s_0 and where each free value (that is the locations not allocated and the fields not presents in s_0) is chosen equal to the corresponding value in s_1'. I called this state the constrained translation: $s_0' = \texttt{constr\_nstate}\ s_1'\ s_0$. I then prove that s_0' reduces to s_1'. This implies the result we are looking for. The reason why we do not simply use this constrained translation as induction hypothesis of the proof (instead of using a simulation relation) is because it cannot be used to compose multiple atomic step. The reason is that in general:

$$\texttt{constr\_nstate (constr\_nstate}\ s_2'\ s_1\texttt{)}\ s_0 \neq \texttt{constr\_nstate}\ s_2'\ s_0$$

2.3 Third step: the translation to Boogie

The last step is a simulation of the Dafny method (with the intermediate semantics) by the `Impl$$` procedure emitted in Boogie. Contrary to the previous steps, this simulation is not done once and forall. Instead we define translation templates that relate Dafny method to Boogie procedures (in a CFG representation). We prove that any translation obtained by combining those templates is correct. Hence one can certify the correctness of an execution of the Dafny verifier by proving that the original Dafny program can be related to its Boogie translation using the templates.
For convenience I defined a wrapper and several utility definitions and lemmas for Boogie in `BoogieWrapper`. In particular, I build reductions with the `cfg_red1` predicate which abstract the block structure of the Boogie CFG because it is not relevant for the translation.
In `TranslationAxioms` we define several declarations that we expect to find in the Boogie program, that is, the types, functions, constants and global variables. Instead of fixing their names, we represent them using datatype values that are then mapped to concrete Boogie identifiers. We also define the carrier type and the type and function interpretations required by the Boogie semantics. We define the carrier type using a datatype that includes the Dafny values via an injection `tr_val`. The Boogie type interpretation is defined directly as a function that map the values of the carrier type to Boogie types. As requested by Boogie, we ensure that any closed type (even undeclared) is inhabited. The carrier type contains some ill-formed values (for instance the ill-typed heaps) that we would like to ignore. Since the well-formedness of the value might depend on the declarations of the Dafny program (for instance the type of the fields) it seems that we cannot remove the unwanted value with a `typedef` and that we would need dependent

types. Since this concept is not present in Isabelle's logic, we instead map the unwanted values to a special `DummyTy` which is assumed to be distinct from any used type.
Because the types of the locations of the heap are encoded as a Boogie function `dtype`, the Boogie function interpretation depends on those types and hence on the trace considered. For this reason it was useful to prove the step from the partial to the complete heap (i.e., the pretty to intermediate semantics) separately, in order to know from the beginning which types to give to the heap locations. Otherwise one would have had to quantify the function interpretation during this simulation step. An alternative would have been to fix a type for each reference value, for instance by using `iref_type` × `nat` as a reference type.

In `Translation`, we define some semantics relations between Dafny and Boogie states, expressions and statements. We prove that the correctness of the `Impl$$` Boogie procedure implies the correctness of the Dafny method implementation under the assumption that the corresponding objects are related by those properties.
Each Dafny variable is represented by a Boogie local variable (the mapping is a parameter of our theory). Our relation `tr_state` between normal states enforces that:

- The value of the global `$Heap` variable is the translation of the heap of the Dafny state and its value in the old state is the translation of the old heap of the Dafny state.
- If a Dafny variable is associated to a value, the corresponding Boogie local contains the translation of this value.

A Dafny expression `d_e` is related to a Boogie expression `b_e` if when `d_e` reduces to a value `v` in a state `d_s`, `b_e` reduces to `tr_val v` in any Boogie state related to `d_s`. This relation does not state anything about the evaluation of `b_e` if `d_e` produces an error. For this reason, the Dafny verifier emits some commands before any expression evaluation in order to check that it does not produce an error. This sequence of command produces a failure if the evaluation of the Dafny expression produces an error and preserves the relation between the Dafny and Boogie states otherwise.
We relate Dafny statements to subsets of the Boogie CFG with a forward simulation strategy. A Dafny statement `stmt` is related to a subset of the Boogie CFG with an entry `in_nd` and two exits `out_nd` and `brk_nd` if for any related `d_s0` and `b_s0`:

- if `stmt` reduces from `d_s0` to `NState d_s1`:

$$\begin{array}{c}\exists\texttt{b\_s1}\text{ related to }\texttt{d\_s1}.\\ (\texttt{in\_nd},\texttt{b\_s0})\xrightarrow{\text{cfg}_*}(\texttt{out\_nd},\texttt{b\_s1})\end{array}\qquad or\qquad (\texttt{in\_nd},\texttt{b\_s0})\xrightarrow{\text{cfg}_*}\texttt{Failure}$$

- the same holds with `brk_nd` if `stmt` reduces to `BreakState d_s1`.
- if `stmt` fails from `d_s0`:

$$(\texttt{in\_nd},\texttt{b\_s0})\xrightarrow{\text{cfg}_*}\texttt{Failure}$$

Hence we can simulate normal reductions with a reduction to `Failure` in Boogie. We use this for the frame inclusion test because it is more liberal in the Dafny semantics than in the Boogie encoding since there are more fields in Boogie. Indeed the Boogie `Field` types also include the `alloc` field and some other fields that we introduce in order to ensure that all `Field` types are inhabited.
A formal simulation relation f for this forward simulation would relate general Dafny states to general Boogie states (at the corresponding locations). f would relate normal Dafny states to the normal Boogie states related with `tr_state` and to `Failure`. The Dafny `FailState` would only br related to `Failure`. The result that has to be proven is the following for a reduction $s_0 \xrightarrow{s} s_1$ in the source semantics [7]:

$$\forall s_0'.\ f(s_0, s_0') \Rightarrow \exists s_1'.\ f(s_1, s_1') \ \wedge\ s_0' \xrightarrow{t*} s_1'$$

Our relation between statements implies this result since there is a reduction $\texttt{Failure} \xrightarrow{*} \texttt{Failure}$.

In `TranslationVerification` we define some templates (as inductive predicates) that relate Dafny and Boogie syntactic constructions. The general structure of those templates follows the internal logic of the Dafny verifier. However we did not reimplement the several heuristics used for simplifications and triggering since the decision made by those heuristics does not affect the correctness of the translation.

Instead our templates can match several translations and our aim is for the actual translation emitted by the verifier to be one of them (proving this would be the main goal of a certification of the translation). We prove that any pair of objects built using those templates are related by our semantics relations and thus is a correct translation. Those proofs are done by induction on the syntactic construction of the translation. The induction hypotheses are the semantic properties defined in `Translation`. We also maintain some syntactic invariants, for instance that the corresponding Boogie local variable of each declared Dafny variable has the appropriate type.

3 Typing

Several type systems are involved:

- The most precise one is the Dafny type system.
- We need to ensure that the translated program is well-typed with respect to the Boogie's type system and that our instantiations (in particular the heap instantiation) satisfy its constraints.
- The `$Is` and `$IsAlloc` predicates encode the type system of Dafny in the translation.

The Dafny type system is used in the front-end semantics for the initialisations and the calls. The Boogie type system is also mentioned indirectly in order to ensure that we will be able to satisfy some type constraints during the simulation to Boogie. Those additional constraints relate the types of the fields of the heap (in the Boogie type system) and can probably be removed by reasoning on the semantics of Dafny.
For the Dafny type system, the type of values are given by `val_typ :: "declarations ⇒ (ref ⇀ iref_type) ⇒ val ⇒ itype ⇒ bool"`. This predicate depends on the types of the allocated locations of the heap (`ref ⇀ iref_type`), indeed a `VRef r` has type `rty` if and only if `r` is `NULL` or the location at reference `r` is allocated and has type `rty`. Since the semantics does not contain any operation that frees or changes the type of an allocated heap location, any pair of heaps that appears in the same execution satisfies an `heap_succ` relation which preserve the type of values.
We prove in `DafnyTypeSafety` the progress and type safety of expressions. The type safety is given in the current heap, inside `Old` expressions one can manipulate values that are not well-typed for the old heap, that is, references to objects that have been allocated during the method execution. This has some consequences on the translation of read operations. Indeed, given a well-typed reference value, we can test the allocation status of the referred location in the current heap by testing the nullity of the reference. However, this does not work for the old heap, and the translation of read operations on the old heap produces an additional assertion.
We prove the type preservation for the pretty semantics and the intermediate semantics and we deduce the type preservation of the front-end semantics using the simulation to the pretty semantics. We also prove that well-typed method body cannot reduces to a `BreakState` and that statements do not modify read-only variables (that is arguments of the method) or locations outside the context `Modifies` frame.
The translation of type parameters does not use the type parameter mechanism of Boogie. Instead a `Ty` type is to used to represent the type parameters as Boogie values. The variables whose Dafny type does not determine the Boogie type are represented using a `Box` type that can store any kind of Boogie value. The `$Box` and `$Unbox` functions allows to store or retrieve a value from a box. Since the `$Unbox` function must be well-typed, it returns an arbitrary value when asked to retrieve a value of a type different from the type of the value stored in the box. Hence, for $\texttt{T} \neq \texttt{S}$:

$$\texttt{\$Unbox<T>(\$Box<S>(v))} \neq \texttt{v}$$

In order to prove the correctness of the translation, one need to use the Dafny type system to prove that each unboxing is done on a box which contains a value of the correct type. As already mentioned for the reading of fields, we also use the Dafny type safety to remove some assertions. Moreover Dafny emits some assumptions that can be freely used by Boogie. For instance, the local variables can be annotated with where clauses which enforce the Dafny type of the value stored. In order to encode the Dafny type system in Boogie, two predicates are defined:

- `$Is` is a typing judgement independent of the heap. Since the types of locations of the heap are fixed during the whole execution, `$Is` depends on them but not on the allocation status of the locations.

- `$IsAlloc` is a typing judgement that depends on the heap. It only takes into account the allocation status of the locations but not their types.

The conjunction of those two typing judgement is equivalent to the typing system of Dafny. They are also used to enforce the type of the fields on the heap. The `$IsGoodHeap` predicates implies (with the type and allocation axioms of the fields emitted by the Dafny verifier) that the fields declared for the type of a location have the correct types. We prove a similar result in the type safety which allows to prove the allocation axioms. However the type axioms enforce the `$Is` type of fields of all locations, even the non-allocated one. In order to prove them, we must ensure that the locations added during the simulation to the intermediate semantics satisfy those axioms.

4 The heap

Our fragment includes operations on the heap: allocations of objects and reading and writing of fields. I've encountered several issues related to the heap:

- The differences between the front-end semantics of Dafny and the heap encoding in Boogie, are the causes of the need of a backward simulation step.
- Providing an instantiation for a Boogie map type with a type parameter is not straightforward since we must produce a correct behaviour even for the types we are not interested in.
- The Dafny axioms and assumptions implies that the map type used to encode heap locations does not satisfy the extensionality axiom.

In order to encode the heap in Boogie, the following declarations are emitted:

```
type ref;
type Field alpha;
const unique alloc: Field bool;
type Heap = [ref]<alpha>[Field alpha]alpha;
```

A heap is encoded as a mapping from references to heap locations. A heap location is again another map from fields to values. This encoding with two map level allows to consider a whole location as a value by using the selection operator only on the first level (`$Heap[r]`), this is used to express the framing at object level. The special field `alloc` is used to store the allocation state of a location. Since Boogie maps are total, a heap contains values even for the locations that are not allocated (the type axioms generated by Dafny even refer to them). The types of the objects on the heap are defined by a separate opaque function `dtype`.
The translation of the allocation operation is affected by this encoding (figure 3). Indeed the translation does not assign its type to the allocated location (since the types are represented with the function `dtype`) and instead assume that the type of the location is the correct one. Moreover it does not change the fields.
Those encoding of the heaps and allocations are the motivations for introducing the intermediate semantics.

```
havoc r;
assume !Heap[r].allocated;
Heap[r].allocated := True;
Heap[r].type := ty;
havoc Heap[r].fields;
```

```
havoc r;
assume !Heap[r].allocated;
Heap[r].allocated := True;
assume Heap[r].type == ty;
```

Figure 3: Pseudocode representations of the semantics of an allocation in the front-end semantics of Dafny (left) and in the Boogie translation (right)

The current mechanization of the Boogie semantics does not support the map types. Instead it desugar the maps by introducing some abstract types and select and store functions. For the heap locations we obtain (after some renaming):

```
type MapHeapLoc;
function select_MapHeapLoc<alpha>(h: MapHeapLoc, f: Field alpha) : alpha;
function store_MapHeapLoc<alpha>(h: MapHeapLoc, f: Field alpha, v: alpha) : MapHeapLoc;
```

To use the result given by a Boogie certificate, we must provide a concrete Isabelle type to instantiate the carrier type for the abstract Boogie values. For the heap location a first attempt was to use a datatype constructor `AHeapLoc (field_a ⇒ tr_a Semantics.val)` (where `tr_a Semantics.val` is the type of Boogie values parametrised with our instantiation of the carrier type). However this raises two issues:

- A general value built using this constructor can map a field of type `Field alpha` to a value which does not belong to the type `alpha`. Indeed `tr_a Semantics.val` is the Isabelle type of all the values used by the Boogie program. Hence we need to remove some ill-typed maps by giving them the type `DummyTy`.
- Since the map is complete, the select function can be applied for any field, that is for any value of type `Field alpha` for some type `alpha`. In particular, since any type should be inhabited in Boogie, there must be a value of type `Field MapHeapLoc`. Hence any well-typed `AHeapLoc` must contain another well-typed `AHeapLoc`. The existence of a well-typed `AHeapLoc` would imply the existence of an infinite sequence of well-typed `AHeapLoc` where each element is a parent of its successor. This is not possible because for an inductive datatype, this relation is well-founded. Hence this datatype does not contain any well typed heap location.

  A possible solution for this issue could be to instantiate the carrier type with a coinductive datatype (see an attempt in branch `snap-heap_coind`). However this brings additional difficulties because we would need to use some (co-)inductive predicates to define the type interpretation whereas we can simply use a function with an inductive datatype.

  Instead, I have chosen to instantiate the heap locations with a partial mapping: `field_a ⇀ tr_a Semantics.val`. By returning an arbitrary value of the appropriate type when asked for a field which is not in the domain of the map, we can provide an interface with select and store functions with the same signatures as for a complete mapping.

The set of partial mappings restricted so that each field is associated to a correct type provides a possible instantiation for the map type of heap locations. The map type of heaps can then be instantiated with complete mappings from references to well-formed heap locations. However I needed to change this instantiation to satisfy the Dafny axioms. Indeed, the Dafny prelude includes the following definitions and axioms:

```
function $IsGoodHeap(Heap): bool;
function $HeapSucc(Heap, Heap): bool;

// AxUpdtHeapSucc
axiom (forall<alpha> h: Heap, r: ref, f: Field alpha, x: alpha :: { update(h, r, f, x) }
  $IsGoodHeap(update(h, r, f, x)) ==> $HeapSucc(h, update(h, r, f, x)));

// AxHeapSuccAlloc
axiom (forall h: Heap, k: Heap :: { $HeapSucc(h,k) }
  $HeapSucc(h,k) ==> (forall o: ref :: { read(k, o, alloc) }
      read(h, o, alloc) ==> read(k, o, alloc)));
```

The `$IsGoodHeap` predicate is freely assumed for any heap that appears in the translation. Since there is no introduction rule for this predicate in the axioms, this is the only way it can be obtained. Its main purpose is to enforce the type of the values on the heap in the encoding of the Dafny type system (with `$Is` and `$IsAlloc`). The `$HeapSucc` predicate is a relation between two heaps,which is satisfied if they appear in that order during a Dafny execution. Since the Dafny semantics does not allow to free a heap location, any location allocated in the first heap is necessarily still allocated in the second one. This is what `AxHeapSuccAlloc` express, in fact, this is the definition of my instantiation of `$HeapSucc`, so the first implication is an equivalence. `$HeapSucc` is freely assumed at some points of the translation (for instance as a free invariant of the loops) but `AxUpdtHeapSucc` also provides a way to introduce it. Under the assumption that the resulting heap satisfies the `$IsGoodHeap` predicate, this axiom states that the result of any update is a successor of the original heap. This is surprising since the update operation in Boogie also includes the updates of the allocation status of the locations with the `alloc` field. The trick that allows to introduce this axiom (without obtaining an inconsistency) is the `$IsGoodHeap` premise. Indeed, since this predicate is only introduced directly for a heap that appears during the translation, it is never assumed on a heap obtained by storing `false` in an `alloc` field. However, if we had an extensionality axiom for the Boogie maps, we could express a heap that appears in the Dafny execution as such update of another heap. Concretely, if we had this axiom, assuming:

`$IsGoodHeap(H)` and `read(H, r, alloc) == False`

We would have, using the extensionality and because `update` and `read` are defined using the store and select operations:

`H == update(update(H, r, alloc, True), r, alloc, False)`

Hence using `AxUpdtHeapSucc`:

`$HeapSucc(update(H, r, alloc, True), H)`

but `read(update(h, r, alloc, True), r, alloc) == True` and `read(H, r, alloc) == False`

which is a contradiction because of `AxHeapSuccAlloc`.
Hence any map instantiations satisfying the extensionality axiom would not satisfy one of the Dafny axioms or assumptions. Our previous attempt was not satisfying the `AxUpdtHeapSucc` axiom. To solve this problem we must ensure that any heap satisfying the `$IsGoodHeap` predicate cannot be expressed as `update(h, r, alloc, false)`. For this purpose we add an additional boolean field to the `AHeapLoc`. This validity marker must be set on any location of a heap satisfying the `$IsGoodHeap` predicate and is unset on a heap obtained by an invalid operation (storing `false` in `alloc`). Since this operation is never done in the Dafny semantics, we are still able to assume the `$IsGoodHeap` predicate for any heap that appears during the execution. This validity marker is not visible explicitly in Boogie, it can only be unset during an assignment and the `$IsGoodHeap` is the only function that reads it.

Conclusion

The proof strategies that I have defined could be used to prove the correctness of executions of the Dafny verifier. I was able to do this proof manually for a modified translation of a simple Dafny program. However, I did not notice during this work any aspect that would make the once-and-forall-certification of a reimplementation of Dafny significantly harder than the verification of executions. Our proofs could also be used in this approach.
This work could be extended by including some aspects of Dafny that are not present in our fragment (for instance the functions). It would also be interesting to prove some properties of the Dafny semantics.

References

[1] My modified version of Dafny, `https://github.com/857b/dafny`.

[2] Arthur Charguéraud. Pretty-Big-Step Semantics. In Matthias Felleisen and Philippa Gardner, editors, *22nd European Symposium on Programming (ESOP)*, Proceedings of the 22nd European Symposium on Programming, Rome, Italy, March 2013. Springer.

[3] Isabelle contributors. *Isabelle*, 2021. Available at `https://isabelle.in.tum.de/index.html`.

[4] K Rustan M Leino. This is boogie 2. *manuscript KRML*, 178(131):9, 2008.

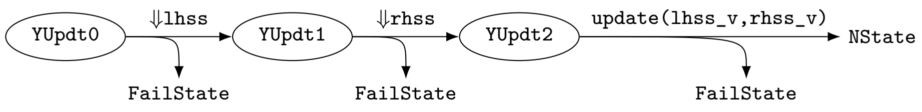
[5] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

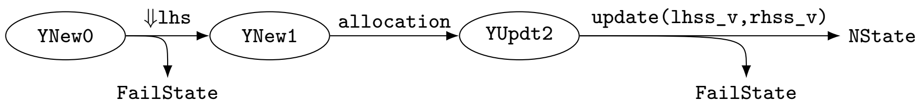
[6] K Rustan M Leino, Richard L Ford, and David R Cok. *Dafny Reference Manual.* January 30, 2021, Available at `https://github.com/dafny-lang/dafny/blob/8af37b56f1ca311cde0b7fa53c2a09794d85b4b4/docs/DafnyRef/out/DafnyRef.pdf`.

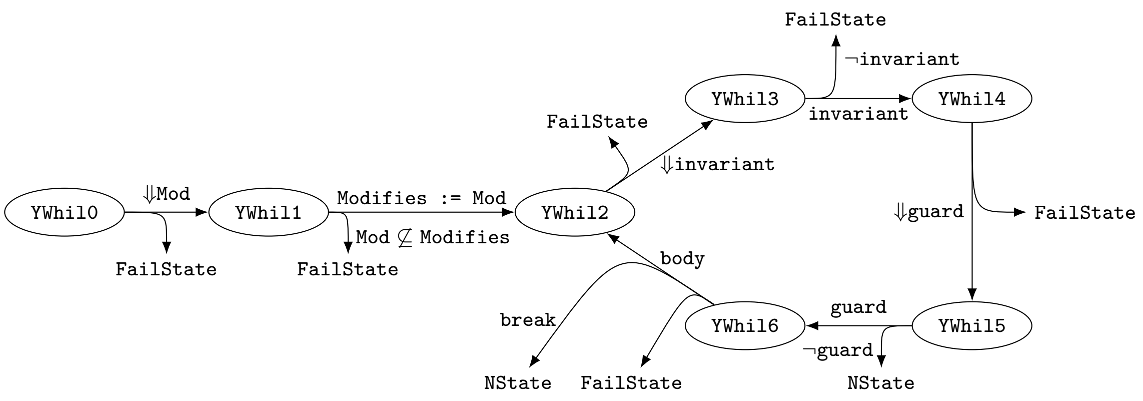
[7] Nancy Lynch and Frits Vaandrager. Forward and backward simulations. *Information and Computation*, 121(2):214–233, 1995.

[8] G. Parthasarathy, P. Müller, and A. J. Summers. Formally validating a practical verification condition generator. In *Computer Aided Verification (CAV)*, 2021. To appear.

Annex: pretty semantics reduction diagrams


YUpdt0
⇓lhss
YUpdt1
⇓rhss
YUpdt2
update(lhss_v,rhss_v)
NState
FailState
FailState
FailState


YNew0
⇓lhs
YNew1
allocation
YUpdt2
update(lhss_v,rhss_v)
NState
FailState
FailState


FailState
¬invariant
YWhil3
invariant
YWhil4
FailState
⇓invariant
YWhil0
⇓Mod
YWhil1
Modifies := Mod
YWhil2
⇓guard
FailState
Mod ⊈ Modifies
FailState
FailState
body
break
YWhil6
guard
YWhil5
¬guard
NState
FailState
NState