Automatically Verifying Typing Constraints for a Data Processing Language

Michael Backes[1,2], Cătălin Hriţcu[1,3], Thorsten Tarrach[1,4,5]

[1]Saarland University, Saarbrücken, Germany [2]MPI-SWS, Saarbrücken, Germany
[3]University of Pennsylvania, Philadelphia, USA
[4]Atomia AB, Västerås, Sweden [5]Troxo DOO, Niš, Serbia

Abstract. In this paper we present a new technique for automatically verifying typing constraints in the setting of Dminor, a first-order data processing language with refinement types and dynamic type-tests. We achieve this by translating Dminor programs into a standard while language and then using a general-purpose verification tool. Our translation generates assertions in the while program that faithfully represent the sophisticated typing constraints in the original program. We use a generic verification condition generator together with an SMT solver to prove statically that these assertions succeed in all executions. We formalise our translation algorithm using an interactive theorem prover and provide a machine-checkable proof of its soundness. We provide a prototype implementation using Boogie and Z3 that can already be used to efficiently verify a large number of test programs.

1 Introduction

Dminor [7] is a first-order data processing language with *refinement types* (types qualified by Boolean expressions) and *dynamic type-tests* (Boolean expressions testing whether a value belongs to a type). The combination of refinement types and dynamic type-tests seems to be very useful in practice [2]. However, the increased expressivity allowed by this combination makes statically type-checking programs very challenging.

In this paper we present a new technique for statically checking the typing constraints in Dminor programs by translating these programs into a standard while language. The sophisticated typing constraints in the original program are faithfully encoded as assertions in the generated program and we use a general-purpose verification tool to show statically that these assertions succeed in all executions. This opens up the possibility to take advantage of the huge amount of proven techniques and ongoing research done on general-purpose verification tools.

We have proved that if all assertions succeed in the translated program then the original Dminor program does not cause typing errors when executed. This proof was done in the Coq interactive theorem prover [5], based on a formalisation of our translation algorithm. We thus show formally that, for the language we are considering, a generic verification tool can check the same properties as a sophisticated type-checker. To the best of our knowledge, this is the first machine-checked proof of a translation to an intermediate verification language (IVL).

Finally, we provide a prototype implementation using Boogie and Z3 that can already be used to verify a large number of test programs and we report on an experimental evaluation against the original Dminor type-checker.

1.1 Related Work

Bierman et al. [7] were the first to study the combination of refinement types and dynamic type-tests. They introduce a first-order functional language called Dminor, which captures the essence of the Microsoft code name M language [2], but which is simple enough to express formally. They show that the combination of refinement types and dynamic type-tests is highly expressive; for instance intersection, union, negation, singleton, dependent sum, variant and algebraic types are all derivable in Dminor. This expressivity comes, however, at a cost: statically type-checking Dminor programs is very challenging, since the type information can be "hidden" deep inside refinements with arbitrary logical structure. For instance intersection types $T\&U$ are encoded in Dminor as refinement types $(x : \mathsf{Any}\ \mathsf{where}\ (x\ \mathsf{in}\ T\ \&\&\ x\ \mathsf{in}\ U))$, where the refinement formula is the boolean conjunction of the results of two dynamic type-tests. Syntax-directed typing rules cannot deal with such "non-structural" types, so Bierman et al. [7] propose a solution based on semantic subtyping. They formulate a semantics in which types are interpreted as first-order logic formulae, subtyping is defined as a valid implication between the semantics of types and they use an SMT solver to discharge such logical formulae efficiently.

The idea of using an SMT solver for type-checking languages with refinement types is quite well established and was used in languages such as SAGE [17], F7 [6], Fine [30] and Dsolve [29]. Bierman et al. [7] show that, in the setting of a first-order language, the SMT solver can play a more central role: They successfully use the SMT solver to check semantic subtyping, not just the refinement constraints. However, while in Dminor [7] subtyping is semantic and checked by the SMT solver, type-checking is still specified by syntax-directed typing rules, and implemented by bidirectional typing rules. In the current work we show that we can achieve very similar results to those of the Dminor type-checker without relying on *any* typing rules, by using the logical semantics of Dminor types directly to generate assertions in a while program, and then verifying the while program using standard verification tools.

Relating type systems and software model-checkers is a topic that has received attention recently from the research community [15, 18, 25]. Our approach is different since we enforce typing constraints using a verification condition generator. Our implementation uses Boogie [20], the generic verification condition generation back-end used by the Verified C Compiler (VCC) [10] and Spec# [4].

There is previous work on integrating verification tools such as Boogie [8] and Why [13] with proof assistants, for the purpose of manually aiding the verification process or proving the correctness of background theories with respect to natural models. However, we are not aware of other machine-checked correctness proofs for translations from surface programming languages into IVLs, even for a language as simple as the one described in this paper. A translation from Java bytecode into Boogie was proved correct in the Mobius project [1, 19], but we are not aware of any mechanised formalisation of this proof.

1.2 Overview

In §2 we provide a brief review of Dminor and in §3 we give a short introduction to our intermediate verification language. §4 and §5 describe our translation algorithm and its implementation. In §6 we compare our work to the Dminor type-checker [7]. Finally, in §7 we conclude and discuss some future work. Further details, our implementation, our Coq formalisation and proofs are all available online at:
`http://www.infsec.cs.uni-saarland.de/projects/dverify`.

2 Review of Dminor (Data Processing Language)

Dminor is a first-order functional language for data processing. We will briefly review this language below; full details are found in the paper by Bierman et al. [7].

Values in Dminor can be simple values (integers, strings, Booleans or null), collections (multi-sets of values) or entities (records). Dminor types include the top type (Any), scalar types (Integer, Text, Logical), collection types ($T*$) and entity types ($\{\ell\colon T\}$). More interestingly, Dminor has *refinement types*: the refinement type ($x : T$ where e) consists of the values x of type T satisfying the arbitrary Boolean expression e.

Syntax of Dminor Expressions:

$e ::=$	Dminor expression
$x \mid k$	variables and scalar constants
$\oplus(e_1, \ldots, e_n)$	primitive operator application
$e_1 ? e_2 : e_3$	conditional (if-then-else)
let $x = e_1$ in e_2	let-expression (x bound in e_2)
e in T	dynamic type-test
$\{\ell_i \Rightarrow e_i\ ^{i \in 1..n}\}$	entity (record with n fields $\ell_i \ldots \ell_n$)
$e.\ell$	selects field ℓ of entity e
$\{v_1, \ldots, v_n\}$	collection (multiset)
$e_1 :: e_2$	adding element e_1 to collection e_2
from x in e_1 let $y = e_2$ accumulate e_3	collection iteration (x, y bound in e_3)
$f(e_1, \ldots, e_n)$	function application

Refinement types can be used to express pre- and postconditions of functions, as shown in the type of removeNulls below, where the postcondition states that the resulting collection has at most as many elements as the original collection.

Refinement type used to encode pre- and postconditions

$e.\mathsf{Count} \triangleq \mathsf{from}\ x\ \mathsf{in}\ e\ \mathsf{let}\ y = 0\ \mathsf{accumulate}\ y + 1$

$\mathsf{NullableInteger} \triangleq x : \mathsf{Any\ where}\ (x\ \mathsf{in\ Integer}\ \|\ x == \mathsf{null})$

$\mathsf{removeNulls}(c : \mathsf{NullableInteger}*) : (x : \mathsf{Integer}*\ \mathsf{where}\ x.\mathsf{Count} \leq c.\mathsf{Count})\ \{$
$\quad \mathsf{from}\ x\ \mathsf{in}\ c\ \mathsf{let}\ y = \{\}\ \mathsf{accumulate}\ ((x \neq \mathsf{null})?(x :: y) : y)$
$\}$

The dynamic type-test expression e in T matches the result of expression e against the type T; it returns **true** if e has type T and **false** otherwise. While dynamic type-tests are useful on their own in a data processing language (e.g. for pattern-matching an XML document against a schema represented as a type [14]), they can also be used inside refinement types, which greatly increases the expressivity of Dminor (e.g. it allows encoding union, intersection, negation types, etc., as seen in the example above, where NullableInteger is an encoded union between type Integer and the singleton type containing only the value null).

Bierman et al. [7] define a big-step operational semantics for Dminor, in which evaluating an expression can return either a value or "error". An error can for instance arise if a non-existing field is selected from an entity. In Dminor such errors are avoided by the type system, but in this work we rule them out using standard verification tools. The type system by Bierman et al. uses semantic subtyping: they formulate a logical semantics (denotational) in which types are interpreted as first-order logic formulae and subtyping is defined as the valid implication between such formulae. More precisely, they define a function $\mathbf{F}[\![T]\!](v)$ that returns a first-order logic formula testing if the value v belongs to a type T. Since (pure) expressions can appear inside refinement types, $\mathbf{F}[\![T]\!]$ is defined by mutual recursion together with two other functions: $\mathbf{R}[\![e]\!]$ returns a first-order logic term denoting the result[1] of an expression e; and $\mathbf{W}[\![T]\!](v)$ a formula that tests if checking whether v is in type T causes an execution error. The reason for the existence of **W** is that **F** is total and has to return a boolean even when evaluating the expression inside a refinement type causes an error. Our translation makes use of the functions **F** and **W** to faithfully encode the typing constraints in Dminor as assertions in the generated while program.

3 Bemol (Intermediate Verification Language)

We define a simple intermediate verification language (IVL) we call Bemol. Bemol is much simplified compared to a generic IVL: the number of language constructs has been reduced and some Dminor-specific constructs that would normally be encoded were added as primitives. We use Bemol to simplify the presentation, the formalisation of our translation and the soundness proof. In our implementation we use Boogie [3, 11, 20] as the IVL and we encode all Bemol constructs that do not have a direct correspondent in Boogie.

3.1 Syntax and Operational Semantics

Bemol is a while language with collections, records, asserts, mutually recursive procedures, variable scoping and evaluation of logical formulae. The syntax of Bemol is separated into two distinct classes: expressions e, which are side-effect free, and commands c, which have side-effects.

Bemol expressions allow basic operations on values, most of which directly correspond to the operations in Dminor. Also the available primitive operators $\oplus$ are the same as in Dminor. The only significant difference is the expression formula f which "magically" evaluates the logical formula f and returns a boolean encoding the validity or invalidity of the formula – such a construct is standard in most IVLs. We use the notation $[\![e]\!]_{st}$ for the evaluation of expression e under state st. In case of a typing error (such as selecting a non-existing field from an entity) $\perp$ is returned.

Syntax of Bemol Expressions:

$e ::=$	Bemol expression
x	variable
v	Dminor value (scalar, collection or entity)
$\oplus(e_1, \ldots, e_n)$	primitive Dminor operator application
$e.\ell$	selects field ℓ of entity e
$e_1[\ell := e_2]$	updates field ℓ in entity e_1 with e_2 (produces new entity)
$e_1 :: e_2$	adds element e_1 to collection e_2 (produces new collection)
$e_1 \backslash \{e_2\}$	removes one instance of e_2 from e_1 (produces new collection)
is_empty e	returns $\mathtt{true}$ if e is the empty collection; $\mathtt{false}$ otherwise
formula f	returns $\mathtt{true}$ if formula f is valid in the current state

[1] Bierman et al. [7] show that $\mathbf{R}[\![e]\!]$ coincides with the big-step operational semantics on pure expressions – i.e., expressions without side-effects such as non-determinism (accumulate) and non-termination (recursive functions).

Syntax of Bemol Commands:

$c ::=$	Bemol command
skip	does nothing
$c_1;\ c_2$	executes c_1 and then c_2
$x\ :=\ e$	assigns the result of e to x
$\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$	conditional
$\mathsf{while}\ e\ \mathsf{inv}\ a\ \mathsf{do}\ c\ \mathsf{end}$	while loop with invariant a
$\mathsf{assert}\ f$	expects that formula f holds, causes error otherwise
$x\ :=\ \mathsf{pick}\ e$	puts an element of e in x (non-deterministic)
$\mathsf{call}\ P$	calls the procedure P
$\mathsf{backup}\ x\ \mathsf{in}\ c$	backs up the current state

Bemol commands manipulate the current global state, which is a total function that maps variables to values. The invariant specified in the while command does not affect evaluation; its only goal is to aid the verification condition generator. The pick command chooses non-deterministically an element from collection e and assigns its value to variable x. The call P command transfers control to procedure P, which will also operate on the same global state. The backup x in c command backs up the current state, executes c and once this is finished restores all variables to their former value except for x. This is useful for simulating a call-stack for procedures, and we also introduce it during the translation to simplify the soundness proof. A similar technique is employed by Nipkow [26] for representing local variables. We use this in our encoding of procedure calls below. The encoding uses an entity to pass multiple arguments.

Encoding of Procedure Calls

$x\ :=\ \mathsf{call}\ \mathsf{P}(e_1, \ldots, e_n) \triangleq$
$\quad\mathsf{backup}\ x\ \mathsf{in}\ ($
$\qquad\mathsf{arg}\ :=\ \{\}; \mathsf{arg}\ :=\ \mathsf{arg}[\ell_1 := e_1]; \ldots; \mathsf{arg}\ :=\ \mathsf{arg}[\ell_n := e_n]; \mathsf{call}\ \mathsf{P}; x\ :=\ \mathsf{ret})$

$\mathsf{procedure}\ \mathsf{P}(x_1, \ldots, x_n)\{\ c\ \} \triangleq \mathsf{proc}\ \mathsf{P}\ \{\ x_1\ :=\ \mathsf{arg}.\ell_1; \ldots; x_n\ :=\ \mathsf{arg}.\ell_n;\ c\ \}$

3.2 Operational Semantics

We define the big-step semantics of Bemol as a relation $st_{init} \xrightarrow{c} r$, where r can be either a final state st_{final} or $\mathbf{Error}$. The only command that can cause an $\mathbf{Error}$ is the assert command; all the other commands simply "bubble up" the errors produced by failed assertions. If an expression evaluates to $\perp$ it will lead to the divergence of the command that contains it, but this does not cause an error.[2]

Operational Semantics

(Eval Skip)
$$\frac{}{st \xrightarrow{\mathsf{skip}} st}$$

(Eval Assign)
$$\frac{[\![e]\!]_{st} = v}{st \xrightarrow{x\ :=\ e} st[x := v]}$$

(Eval Seq)
$$\frac{st \xrightarrow{c_1} st' \quad st' \xrightarrow{c_2} r}{st \xrightarrow{c_1;\ c_2} r}$$

(Eval Seq Error)
$$\frac{st \xrightarrow{c_1} \mathbf{Error}}{st \xrightarrow{c_1;\ c_2} \mathbf{Error}}$$

[2] Since we only reason about partial correctness, diverging programs are considered correct. This makes the assumptions on our encoding of Bemol into Boogie be minimal: we only assume that the asserts and successful evaluations of the other commands are properly encoded in Boogie. In §4 we prove that we add enough asserts to capture all errors in the original Dminor program, even under these conservative assumptions we make in the Bemol semantics.

(Eval If True)
$$\frac{[\![e]\!]_{st} = \mathbf{true} \quad st \xrightarrow{c_1} r}{st \xrightarrow{\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2} r}$$

(Eval If False)
$$\frac{[\![e]\!]_{st} = \mathbf{false} \quad st \xrightarrow{c_2} r}{st \xrightarrow{\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2} r}$$

(Eval While End)
$$\frac{[\![e]\!]_{st} = \mathbf{false}}{st \xrightarrow{\mathsf{while}\ e\ \mathsf{inv}\ a\ \mathsf{do}\ c\ \mathsf{end}} st}$$

(Eval While Loop)
$$\frac{[\![e]\!]_{st} = \mathbf{true} \quad st \xrightarrow{c} st' \quad st' \xrightarrow{\mathsf{while}\ e\ \mathsf{inv}\ a\ \mathsf{do}\ c\ \mathsf{end}} r}{st \xrightarrow{\mathsf{while}\ e\ \mathsf{inv}\ a\ \mathsf{do}\ c\ \mathsf{end}} r}$$

(Eval While Error)
$$\frac{[\![e]\!]_{st} = \mathbf{true} \quad st \xrightarrow{c} \mathbf{Error}}{st \xrightarrow{\mathsf{while}\ e\ \mathsf{inv}\ a\ \mathsf{do}\ c\ \mathsf{end}} \mathbf{Error}}$$

(Eval Assert)
$$\frac{[\![\mathsf{formula}\ f]\!]_{st} = \mathbf{true}}{st \xrightarrow{\mathsf{assert}\ f} st}$$

(Eval Assert Error)
$$\frac{[\![\mathsf{formula}\ f]\!]_{st} = \mathbf{false}}{st \xrightarrow{\mathsf{assert}\ f} \mathbf{Error}}$$

(Eval Pick)
$$\frac{v \in [\![e]\!]_{st}}{st \xrightarrow{x\ :=\ \mathsf{pick}\ e} st[x := v]}$$

(Eval Call)
$$\frac{st \xrightarrow{c} r \quad \text{given } P\{c\}}{st \xrightarrow{\mathsf{call}\ P} r}$$

(Eval Backup)
$$\frac{st \xrightarrow{c} st'}{st \xrightarrow{\mathsf{backup}\ x\ \mathsf{in}\ c} st[x := st'x]}$$

(Eval Backup Error)
$$\frac{st \xrightarrow{c} \mathbf{Error}}{st \xrightarrow{\mathsf{backup}\ x\ \mathsf{in}\ c} \mathbf{Error}}$$

3.3 Hoare Logic and Verification Condition Generation

We define a Hoare logic for our commands, based on the Software Foundations lecture notes [27] and the ideas of Nipkow [26].

Definition 1 (Hoare Triple). *We say that a Hoare triple $\models \{P\}\, c\, \{Q\}$ holds semantically if $\forall st\ r.\ st \xrightarrow{c} r \implies \forall z.\ P\ z\ st = \mathbf{true} \implies (\exists st'.\ r = st' \wedge Q\ z\ st' = \mathbf{true})$.*

By requiring that the result of the command is not **Error** but an actual state st' we ensure that correct programs do not cause assertions to fail. The meta-variable z is an addition to the traditional Hoare triple and models auxiliary variables, which need to be made explicit in the presence of recursive procedures. Our treatment of auxiliary variables and procedures follows the one of Nipkow [26], who formalises an idea by Morris [24] and Kleymann [16]. We also use Nipkow's definition of extended Hoare triples and Hoare judgements, which are needed for recursive and mutually recursive procedures respectively.

Definition 2 (Extended Hoare Triple). *We say that an extended Hoare triple $C \models \{P\}\, c\, \{Q\}$ holds semantically if and only if $(C$ valid $\implies\ \models \{P\}\, c\, \{Q\})$, where C is valid means $\forall P\ c\ Q.\ \{P\}c\{Q\} \in C \implies\ \models \{P\}\, c\, \{Q\}$.*

Definition 3 (Hoare Judgement). *We say that a Hoare judgement $C_1 \models C_2$ holds semantically if and only if C_1 valid $\implies$ C_2 valid.*

We give a complete list of the Hoare rules for our commands. Except for backup, pick and assert they are the same as in Nipkow's work [26].

Hoare Rules for Bemol

(Hoare Assign)
$$\frac{}{C \models \{Q\{v/x\}\}\, x\ :=\ v\, \{Q\}}$$

(Hoare Sequence)
$$\frac{C \models \{P\}\, c_1\, \{Q\} \quad C \models \{Q\}\, c_2\, \{R\}}{C \models \{P\}\, c_1;\ c_2\, \{R\}}$$

(Hoare If)
$$\frac{C \models \{\lambda z\ st.\ P\ z\ st \wedge [\![e]\!]_{st}\}\, c_1\, \{Q\} \quad C \models \{\lambda z\ st.\ P\ z\ st \wedge \neg[\![e]\!]_{st}\}\, c_2\, \{Q\}}{C \models \{P\}\ \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \{Q\}}$$

(Hoare Skip)

$$\frac{}{C \models \{Q\}\, \mathsf{skip}\, \{Q\}}$$

(Hoare While)

$$\frac{C \models \{\lambda z\ st.\ P\ z\ st \wedge [\![e]\!]_{st}\}\, c\, \{P\}}{C \models \{P\}\, \mathsf{while}\ e\ \mathsf{inv}\ P\ \mathsf{do}\ c\ \mathsf{end}\, \{\lambda z\ st.\ P\ z\ st \wedge \neg [\![e]\!]_{st}\}}$$

(Hoare Context)

$$\frac{\{P\}\, c\, \{Q\} \in C}{C \models \{P\}\, c\, \{Q\}}$$

(Hoare Pick)

$$\frac{}{C \models \{\lambda z\ st.\ \forall v \in [\![e]\!]_{st}, P\{v/x\}\ z\ st\}\, x\ :=\ \mathsf{pick}\ e\, \{P\}}$$

(Hoare Consequence)

$$\frac{C \models \{P'\}\, c\, \{Q'\} \qquad \forall st\ st'.\ (\forall z.\ P'\ z\ st \implies Q'\ z\ st') \implies (\forall z.\ P\ z\ st \implies Q\ z\ st')}{C \models \{P\}\, c\, \{Q\}}$$

(Hoare Triple)

$$\frac{\forall P\ c\ Q.\ \{P\}\, c\, \{Q\} \in C_2 \implies C_1 \models \{P\}\, \mathsf{c}\, \{Q\}}{C_1 \models C_2}$$

(Hoare Call MutRec)

$$\frac{\forall P\ c\ Q.\ \{P\}\, c\, \{Q\} \in C_2 \implies \exists S.\ \ c = \mathsf{call}\ S \qquad \forall P\ Q\ S.\ \ \{P\}\, \mathsf{call}\ S\, \{Q\} \in C_2 \implies C_1 \cup C_2 \models \{P\}\, c\, \{Q\} \quad \text{given } S\{c\}}{C_1 \models C_2}$$

(Hoare Assert)

$$\frac{}{C \models \{Q \wedge a\}\, \mathsf{assert}\ a\, \{Q\}}$$

(Hoare Call Simple)

$$\frac{\{P\}\, \mathsf{call}\ S\, \{Q\} :: C \models \{P\}\, c\, \{Q\} \quad \text{given } S\{c\}}{C \models \{P\}\, \mathsf{call}\ S\, \{Q\}}$$

(Hoare Backup)

$$\frac{\forall st'.\ C \models \{\lambda z\ st.\ P\ z\ st \wedge st' = st\}\, c\, \{\lambda z\ st.\ Q\{st\ x/x\}\ z\ st'\}}{C \models \{P\}\, \mathsf{backup}\ x\ \mathsf{in}\ c\, \{Q\}}$$

The $\mathsf{backup}\ x\ \mathsf{in}\ c$ command requires that the Hoare triple for c has the same state for the pre- and postcondition, except for variable x which is updated. We "transfer" the state from the pre- to the postcondition by quantifying over a new state st' that we require to be equal to the state in the precondition.

For our semantics of the Hoare triples it is possible to define a weakest precondition, but not a strongest postcondition function. This is because if c evaluates to **Error** no post-condition is strong enough to make the triple valid. Corresponding to the Hoare rules, we define a verification condition generator ($\mathsf{VCgen}\ c\ Q$), which takes a command c and a postcondition Q as arguments and generates a precondition. We have proved that this is sound; however, the VCgen is not guaranteed to return the weakest precondition, because the used loop invariants are not necessarily the best. The soundness proof of the VCgen crucially relies on the soundness of the Hoare logic rules above.

More importantly for our application, we have proved as a corollary that the programs deemed correct by our VCgen do not cause errors when executed.

Theorem 1 (Soundness of VCgen).
If $\mathsf{VCgen}\ c\ Q$ *returns a valid formula, then* $\nexists st.\ st \xrightarrow{c} \mathbf{Error}$.

4 Translation from Dminor to Bemol

4.1 Translation Function

Our translation algorithm is a function $\langle\!\langle\, e \,\rangle\!\rangle_{x}$ that takes a Dminor program and a variable name x as input and outputs a Bemol program. The variable x is where the generated Bemol program should store the result after it executes. Variables beginning with z are freshly chosen, meaning no variable with that name exists in the program.

Translation from Dminor to Bemol

$$
\begin{array}{lcl}
\langle\!\langle\, e \,\rangle\!\rangle_{x} & = & \mathsf{backup}\ x\ \mathsf{in}\ \langle e\rangle_x \\
\langle x\rangle_{\mathsf{out}} & = & \mathsf{out}\ :=\ x \\
\langle v\rangle_{\mathsf{out}} & = & \mathsf{out}\ :=\ v \\
\langle \oplus(e_1,\ldots,e_n)\rangle_{\mathsf{out}} & = & \langle\!\langle\, e_1 \,\rangle\!\rangle_{z_1};\ldots;\ \langle\!\langle\, e_n \,\rangle\!\rangle_{z_n}; \\
\text{where operands of } \oplus & & \mathsf{assert}\ \mathbf{F}[\![T_1]\!](z_1);\ldots;\mathsf{assert}\ \mathbf{F}[\![T_n]\!](z_n); \\
\text{need to have types } T_1,\ldots,T_n & & \mathsf{out}\ :=\ \oplus(z_1,\ldots,z_n) \\
\langle e_1?e_2 : e_3\rangle_{\mathsf{out}} & = & \langle\!\langle\, e_1 \,\rangle\!\rangle_{z_1};\ \mathsf{assert}\ \mathbf{F}[\![\mathsf{Logical}]\!](z_1); \\
 & & \mathsf{if}\ z_1\ \mathsf{then}\ \langle\!\langle\, e_2 \,\rangle\!\rangle_{\mathsf{out}}\ \mathsf{else}\ \langle\!\langle\, e_3 \,\rangle\!\rangle_{\mathsf{out}} \\
\langle \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\rangle_{\mathsf{out}} & = & \langle\!\langle\, e_1 \,\rangle\!\rangle_{x};\ \langle\!\langle\, e_2 \,\rangle\!\rangle_{\mathsf{out}} \\
\langle e\ \mathsf{in}\ T\rangle_{\mathsf{out}} & = & \langle\!\langle\, e \,\rangle\!\rangle_{z_1};\ \mathsf{assert}\ (\neg\mathbf{W}[\![T]\!](z_1)); \\
 & & \mathsf{out}\ :=\ \mathsf{formula}\ \mathbf{F}[\![T]\!](z_1) \\
\langle \{\ell_i \Rightarrow e_i\ ^{i\in 1..n}\}\rangle_{\mathsf{out}} & = & \langle\!\langle\, e_1 \,\rangle\!\rangle_{z_1};\ldots;\ \langle\!\langle\, e_n \,\rangle\!\rangle_{z_n};\ \mathsf{out}\ :=\ \{\}; \\
 & & \mathsf{out}\ :=\ \mathsf{out}[\ell_1 := z_1];\ldots;\mathsf{out}\ :=\ \mathsf{out}[\ell_n := z_n] \\
\langle e.\ell\rangle_{\mathsf{out}} & = & \langle\!\langle\, e \,\rangle\!\rangle_{z_1};\ \mathsf{assert}\ \mathbf{F}[\![\{\ell : \mathsf{Any}\}]\!](z_1); \\
 & & \mathsf{out}\ :=\ z_1.\ell \\
\langle \{v_1,\ldots,v_n\}\rangle_{\mathsf{out}} & = & \mathsf{out}\ :=\ \{v_1,\ldots,v_n\} \\
\langle e_1 :: e_2\rangle_{\mathsf{out}} & = & \langle\!\langle\, e_1 \,\rangle\!\rangle_{z_1};\ \langle\!\langle\, e_2 \,\rangle\!\rangle_{z_2};\ \mathsf{assert}\ \mathbf{F}[\![\mathsf{Any}*]\!](z_2); \\
 & & \mathsf{out}\ :=\ z_1 :: z_2 \\
\langle \mathsf{from}\ x\ \mathsf{in}\ e_1 & = & \langle\!\langle\, e_1 \,\rangle\!\rangle_{z_c};\ \langle\!\langle\, e_2 \,\rangle\!\rangle_{y}; \\
\quad \mathsf{let}\ y = e_2 & & \mathsf{assert}\ \mathbf{F}[\![\mathsf{Any}*]\!](z_c); \\
\quad \mathsf{accumulate}\ e_3\rangle_{\mathsf{out}} & & \mathsf{while}\ !(\mathsf{is\_empty}\ z_c)\ \mathsf{inv}\ i(z_c,\ldots)\ \mathsf{do} \\
 & & \quad x\ :=\ \mathsf{pick}\ z_c; \\
 & & \quad z_c\ :=\ z_c\backslash\{x\}; \\
 & & \quad \langle\!\langle\, e_3 \,\rangle\!\rangle_{y} \\
 & & \mathsf{end}; \\
 & & \mathsf{out}\ :=\ y \\
\langle f(e_1,\ldots,e_n)\rangle_{\mathsf{out}} & = & \langle\!\langle\, e_1 \,\rangle\!\rangle_{z_1};\ldots;\ \langle\!\langle\, e_n \,\rangle\!\rangle_{z_n}; \\
 & & \mathsf{out}\ :=\ \mathsf{call}\ f(z_1,\ldots,z_n)
\end{array}
$$

Our translation directly uses function **F** of the Dminor semantics of types [7] to generate logical formulae that check whether a value has a certain Dminor type or not. These formulae are used in asserts to make sure that the variables involved in a command have the right types before actually executing the command. For type-tests we first use an assert $(\neg\mathbf{W}[\![T]\!](z_1))$ to make sure that the type-test itself does not cause an error in Dminor, before actually checking whether z_1 has type T using **F**. To translate procedure calls we use the encoding of call given in §3.1. In our translation of accumulate the loop invariant i needs to be provided as an input, either directly in the translated code, or in the original Dminor program as a type annotation T on the accumulator (as required by the Dminor type-checker). In the later case we can use $\mathbf{F}[\![T]\!]$ to obtain an invariant; however, below we show an example where the necessary invariant cannot be expressed this way. In §5.1 we discuss how we use the Dminor infrastructure for automatically inferring invariants for a special class of accumulate expressions corresponding to LINQ queries [23].

4.2 Examples

In the examples below we consider out to be the variable where the result is put. In Example 1 we show how the removeNulls example from §2 is translated to a while loop that picks and removes elements from the collection until it is empty.

Example 1: Accumulate filtering null **values**

```
removeNulls($c$ : NullableInteger$*$) :
   ($x$ : (Integer$*$) where ($x$.Count $\leq$ $c$.Count))
{
   from $x$ in $c$ let $y = \{\}$
       accumulate (($x \neq$ null)?($x$ :: $y$) : $y$)
}
```

```
procedure removeNulls($c$) {
   assert $\mathbf{F}[\![$NullableInteger$*]\!](c)$;
   $y$ := {};
   $c'$ := $c$;
   while !is_empty $c'$ inv $i(c, c', y)$ do
      $x$ := pick $c'$;
      $c'$ := $c' \setminus \{x\}$;
      if $x \neq$ null then
         $y$ := $x$ :: $y$
      else
         $y$ := $y$
   end;
   ret := $y$;
   assert ($\mathbf{F}$ [[$x$ : Integer$*$
      where $y$.Count $\leq$ $c$.Count) ret]]
}
```

$i(c, c', y) \triangleq \mathbf{F}[\![y : (\text{Integer}*) \text{ where } (c'.\text{Count} + y.\text{Count} \leq c.\text{Count})]\!]\ y$

The loop invariant specifies that the sum of the number of elements in the intermediate collection c' and the resulting collection y is less than or equal to the number of elements in the original collection c. It is not sufficient for the invariant to just reason over y and c as this would be too weak. In this case the invariant is provided by hand on the generated code because this loop invariant is not expressible as a Dminor type. Loop invariant inference on the Dminor side is in general deemed to fail for global properties of collections. Our implementation successfully verifies this example with the provided invariant. In the future we hope to infer such invariants automatically using the Boogie infrastructure for this task.

As seen in Example 2 for type-tests we first use an assert to check that the type-test does not cause a typing error and then perform the actual type-test which returns a Logical. Note that **F** is total and would also return a value on a wrongly typed argument.

Example 2: Type-test

```
$x$ in ($y$ : Integer where $y > 5$)
```

```
assert ($\neg(\mathbf{W}[\![y$ : Integer where $y > 5]\!]\ x)$);
out := formula ($\mathbf{F}[\![y$ : Integer where $y > 5]\!]\ x$)
```

For illustration, we expand $\mathbf{W}$ and $\mathbf{F}$ in the example above; please see the paper by Bierman et al. [7] for the precise definition of these functions of the logical semantics.

$$
\begin{array}{l}
\quad \mathbf{W}[\![y : \mathsf{Integer\ where}\ y > 5]\!]\ x \\
\models \mathbf{W}[\![\mathsf{Integer}]\!]\ x \vee \mathsf{let}\ y = x\ \mathsf{in} \\
\qquad \neg(\mathbf{R}[\![y > 5]\!] = \mathbf{Return}(\mathbf{false}) \vee \mathbf{R}[\![y > 5]\!] = \mathbf{Return}(\mathbf{true})) \\
\models \mathbf{false} \vee \neg((\mathsf{if}\ \mathbf{F}[\![\mathsf{Integer}]\!]\ x\ \mathsf{then}\ \mathbf{Return}(x > 5)\ \mathsf{else}\ \mathbf{Error}) = \mathbf{Return}(\mathbf{false}) \\
\qquad\qquad\quad \vee\ (\mathsf{if}\ \mathbf{F}[\![\mathsf{Integer}]\!]\ x\ \mathsf{then}\ \mathbf{Return}(x > 5)\ \mathsf{else}\ \mathbf{Error}) = \mathbf{Return}(\mathbf{true})) \\
\models \neg\mathbf{F}[\![\mathsf{Integer}]\!]\ x \models \neg(\mathsf{In\_Integer}\ x)
\end{array}
$$

$$
\begin{array}{l}
\quad \mathbf{F}[\![y : \mathsf{Integer\ where}\ y > 5]\!]\ x \\
\models \mathbf{F}[\![\mathsf{Integer}]\!]\ x \wedge \mathsf{let}\ y = x\ \mathsf{in}\ \mathbf{R}[\![y > 5]\!] = \mathbf{Return}(\mathbf{true}) \\
\models \mathsf{In\_Integer}\ x \wedge (\mathsf{if\ In\_Integer}\ x\ \mathsf{then}\ \mathbf{Return}(x > 5)\ \mathsf{else}\ \mathbf{Error}) = \mathbf{Return}(\mathbf{true}) \\
\models \mathsf{In\_Integer}\ x \wedge x > 5
\end{array}
$$

In case x is not an integer the formula In_Integer $x \wedge x > 5$ is logically equivalent to **false**. Our translation asserts that x is an integer before calling formula in order to match the semantics of Dminor, in which $x > 5$ causes an error when x is not an integer.

We construct another example using type-tests where our technique is more complete than the Dminor type system.

Example 3: Valid type-test

4 in (x : Any where $x > 5$)	assert ($\neg\mathbf{W}[\![x : \mathsf{Any\ where}\ x > 5]\!](4)$);
	out := formula ($\mathbf{F}[\![x : \mathsf{Any\ where}\ x > 5]\!]\ 4$)

The Dminor type system rejects Example 3 as ill-typed because x is typed as Any, which is not a valid type for an operand of the greater operator. The big-step semantics of Dminor, however, evaluates this expression successfully to **false** because the x always evaluates to 4, which is a valid operand of the greater operator. The translated program is accepted by Boogie, since our translation aims to be complete with respect to the operational semantics, whereas Dminor implements an inherently incomplete type system.

4.3 Soundness

We have proved in Coq that if a Dminor program e can evaluate to **Error**[3], then the translated program $\langle\!\langle\, e \,\rangle\!\rangle_x$ can evaluate to **Error** in Bemol. The contrapositive of this is: if the translated program cannot evaluate to **Error**, then the original Dminor program cannot evaluate to **Error** either. We have proved this theorem in Coq by induction over the big-step semantics of Dminor $\Downarrow$.

Theorem 2 (Soundness of the Translation). *If* $e \Downarrow \mathbf{Error}$ *then* $\forall st.\ st \xrightarrow{\langle\!\langle\, e \,\rangle\!\rangle_x} \mathbf{Error}$.

As an immediate consequence of Theorem 1 and Theorem 2 we obtain the soundness of our whole technique.

Corollary 1 (Soundness). *If* VCgen $\langle\!\langle\, e \,\rangle\!\rangle_x$ true *is a valid formula, then* $e \not\Downarrow \mathbf{Error}$.

[3] Because of non-determinism a program could evaluate to **Error** only in some of the possible executions.

4.4 Formalisation and Machine-checked Proofs

We have proved Theorem 2 in Coq, by mutual induction together with Lemma 1.

Lemma 1. *If* $e \Downarrow v$ *then* $\forall st.\ \exists st'.\ st \xrightarrow{\langle\!\langle e \rangle\!\rangle_x} st'$ *and* $st'x = v$*.*

To prove this we require three additional assumptions: The expression we want to translate must not contain impure refinements, none of the functions contains impure refinements and the only free variable a function may have is the argument[4]. An impure refinement is when an impure expression (possibly non-deterministic or non-terminating) is used in a refinement type.

We prove this theorem by induction on the Dminor big-step semantics [7], which gives us 42 cases. In each case we have to prove that the generated Bemol code evaluates to the same result (value or error) as the original Dminor expression. On the Bemol side we use the big-step semantics we have defined in §3.2.

The first step in the proof is always to remove the backup command that is added by every translation step. For this we use two lemmas, one for the error case and one for successful evaluation. In the case of successful evaluation only the output variable out changed after executing the backup command. This fact significantly simplifies the proof and is the main reason we add a backup command at every step of the translation.

Lemma 2 (Backup Error). $st \xrightarrow{\mathsf{backup}\ x\ \mathsf{in}\ c}$ **Error** *if and only if* $st \xrightarrow{c}$ **Error**

Lemma 3 (Backup). *If* $st \xrightarrow{c} st'$ *and* $st'\ x = v$ *then* $st \xrightarrow{\mathsf{backup}\ x\ \mathsf{in}\ c} st[x := v]$

In our proof we had to strengthen the induction hypotheses in several different ways. First, as already mentioned above, we needed to prove Theorem 2 together with Lemma 1 by mutual induction. Second, we needed to generalize the statements of Theorem 2 and Lemma 1 to open expressions. Since the big-step semantics of Dminor is only defined for closed expressions we needed to substitute the free variables with the values from the Bemol state before evaluating Dminor expressions. Finally, we also needed to strengthen the inner induction hypotheses of a number of cases, such as the accumulate and the entity creation cases.

Our formal development consists of 5000 lines of Coq and our proofs are done in full detail. Our development is made on top of the Dminor formalisation consisting of 4000 lines [7], which makes the total size of the formal development approach 9000 lines of Coq. The soundness proof alone consists of 1300 lines of Coq code and the Coq proof checker takes more than $2\frac{1}{2}$ minutes to check the proof. Three custom Coq tactics were defined to solve steps that are commonly used in the translation proof.

5 Implementation

Our implementation is called DVerify and translates a Dminor program into a Boogie program. DVerify is written in F# 2.0 [22] and consists of more than 1200 lines of code, as well as a 700 line axiomatisation that defines the Dminor types and functions in Boogie. The Boogie tool then takes the translated Boogie program as input and outputs either an error message that describes points in the program where certain postconditions or assertions may not hold [21] or otherwise prints a message indicating that the program has been verified successfully.

[4] For a formal statement see the third author's Master's thesis [31], or the Coq formalization, where Theorem 2 is named `translation_closed_sound`.

5.1 High-level Overview

The heart of our translation algorithm consists of a recursive function that goes over a Dminor expression and translates it into Boogie code (corresponding to the $\langle\!\langle\, e \,\rangle\!\rangle_x$ function in §4.1). This function is called once per Dminor function and produces a Boogie procedure. Types in Dminor are translated into Boogie function symbols returning bool, using another recursive function in our implementation.

Our translation uses the type annotations on accumulate expressions in the original Dminor program to generate invariants for while loops, so that very often the user does not have to provide loop invariants for the generated Boogie program. However, as illustrated by Example 1, there are also cases for which the invariant needed to verify the program is not expressible as a Dminor type. Such loop invariants are completely out of reach for the Dminor type system, and currently can be provided manually in DVerify. In the future we intend to infer such invariants automatically using the Boogie infrastructure for this task.

The Dminor implementation allows for one more construct to define a loop, a from-where-select as in LINQ [23]. In theory from-where-select can be encoded using accumulate, but in the Dminor implementation it is considered primitive in the interest of efficiency and to reduce the type annotation burden. Since from-where-select does not carry a type annotation, we have to find one during translation so that we can use it as a loop invariant. For that we use a modified version of the type-synthesis from Dminor that does not call the type-checking algorithm and therefore never fails to synthesise a type for an expression.

We use the Dminor implementation as a library so that we do not have to reimplement existing functionality. This is mainly the parser for Dminor files, the purity checking and a weak form of type-synthesis for from-where-select.

5.2 Axiomatisation

Our implementation also comprises an axiomatisation of Dminor values and functions in Boogie. This is necessary because Boogie as such understands only two sorts, bool and int, whereas Dminor and Bemol have a number of primitive and composite values, such as collections and entities. Our axiomatisation is similar to the axiomatisation the Dminor type-checker feeds to Z3 [7]. In Dminor this axiomatisation is written in SMT-LIB 1.2 syntax [28] and directly fed to Z3 with the proof obligation. Our axiomatisation is in the Boogie language and Boogie translates it to Simplify syntax [12] and feeds it to Z3 along with the verification conditions it generates. Dminor makes heavy usage of the theories Z3 offers, such as extensional arrays and datatypes. We use the weak arrays provided by Boogie by default and encode datatypes by hand.

6 Comparison between Dminor and DVerify

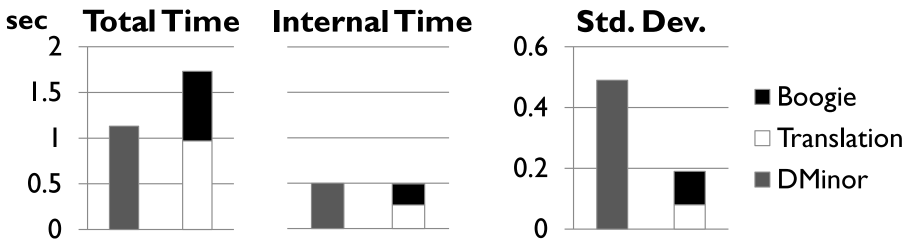
We have tested our implementation against Dminor 0.1.1 from September 2010. Microsoft Research gave us access to their Dminor test suite that contains 109 sample programs. Out of these 109 tests 76 are well-typed Dminor programs and 33 are ill-typed. Out of the 76 well-typed programs, the Dminor type-checker cannot verify 10 tests because of its imprecision.

As shown in Table 1, from the 66 cases on which Dminor succeeds, DVerify manages to verify 62 as correct. Out of the 33 that Dminor rejects, DVerify rejects 31. The other two are correct operationally, but are ill-typed with respect to the (inherently incomplete)

Table 1 Precision Comparison

	well-typed	ill-typed
Test suite	76	33
DMinor accepts	66	0
DVerify accepts	62	2 (correct programs)

Chart 1 Speed Comparison (average times for 66 well-typed samples)



Dminor type system. Overall this means that DVerify succeeds on 94% of the cases Dminor succeeds on and is able to verify two correct programs Dminor cannot verify. For the 4 correct programs that DVerify cannot verify the most common problem is that type-synthesis generates too complicated loop invariants and Z3 cannot handle the resulting proof obligations. Giving explicit type annotations on the Dminor side (instead of relying on Dminor type-synthesis) makes DVerify also accept these programs.

In order to compare efficiency, we first measured the overall wall-clock time that is needed by the two tools, which includes the time the operating system requires to start the process. Because we are dealing with a large number of small test files and both tools are managed .NET assemblies, initialisation dominates the total running times of both tools. Since initialisation is a constant factor that becomes negligible on bigger examples, we also measured the time excluding initialisation and parsing, which we call "internal time". Chart 1 shows both times (averaged over the 62 well-typed samples accepted by both tools) on a 2.1 GHz laptop. The internal time is 0.5s on average for both Dminor and DVerify, which means that both tools are very efficient and that our combination of a translation and an off-the-shelf verification condition generator matches the average speed of a well-optimised type-checker on its own test suite. One should still keep in mind that all examples in this test suite are relatively small, the biggest one consisting of 176 lines.

7 Conclusion

In this paper we have presented a new technique for statically checking the typing constraints in Dminor programs by translating these programs into a standard while language and then using a general-purpose verification tool. We have formalised our translation algorithm using an interactive theorem prover and provide a machine-checkable proof of its

Table 2 Qualitative Comparison of Dminor and DVerify

Area	**Our verification approach (DVerify)**	**Type-checking approach (DMinor)**
Verification cond. generation	Weakest precondition	Bidirectional type-checking (type synthesis ≈ strongest postcondition)
Formulae discharged	One per postcondition/assertion (larger, but less obligations)	One per subtyping test (smaller, but more obligations)
Backend	Boogie + SMT-Solver (Z3)	SMT-Solver (Z3)
Loop invariants	In principle Boogie could infer some (even for accumulates) (even for global properties)	For from-where-select (but not for accumulates) (but not for global properties)
Speed	similar	similar
Precision (practise)	similar	similar
Completeness (theory)	possibly better	possibly worse (type system)
Theories	equality, integers, datatypes, weak arrays	equality, integers, extensional arrays and native encoding of datatypes

soundness. We also provide a prototype implementation using Boogie and Z3, which can already be used to verify a large number of test programs and which is able to match and on some examples even surpass the precision and efficiency of the Dminor type-checker.

Future Work

Using a general verification tool for checking the types of Dminor programs should allow us to increase the expressivity of the Dminor language more easily. For example, adding support for *mutable state* would be easy in DVerify: Bemol already supports state, moreover Boogie is used mainly for imperative programming languages [9]. An interesting consequence is that it should be easier to support strong updates in DVerify (i.e. updates that change the type of variables), which is usually quite hard to achieve with a type-checker.

Another very interesting extension is *inferring loop invariants*. Dminor requires that each accumulate expression is annotated with a type for the accumulator which constitutes the invariant of the loop, whereas Boogie has build-in support for abstract interpretation for automatically inferring such invariants [3]. While the invariant inference support in Boogie seems currently very much focused on integer domains, it seems possible to extend it to include support for our Dminor types.

Completeness. A theoretical goal would be to prove the completeness of our technique, rather than just soundness. Completeness would ensure that if our VCgen for Bemol generates an invalid formula, then the original program indeed evaluates to an error. This would guarantee that the only source of false positives (i.e., programs that are rejected by our technique, but are actually correct with respect to the Dminor big-step semantics) is the tool used to discharge the verification conditions (i.e., Z3). A crucial step in this direction would be to show the translation complete.

Claim (Completeness of translation). If $\forall st.\ st \xrightarrow{\langle\!\langle e \rangle\!\rangle_x} \mathbf{Error}$ then $e \Downarrow \mathbf{Error}$ and if $\forall st, st'.\ st \xrightarrow{\langle\!\langle e \rangle\!\rangle_x} st'$ then $e \Downarrow st'\ x$.

As for soundness, the completeness of the translation can probably be combined with the completeness of the Hoare logic. We expect our Hoare logic to be complete because Nipkow proved completeness for a similar set of Hoare rules [26]. The verification condition generator is, however, inherently incomplete, because of the user-provided annotations for loop invariants and procedure pre- and postconditions. However, for loop-and-procedure-free programs a completeness proof should be possible even for the verification condition generator. Even more, one should be able to prove the *expressive completeness* of the verification condition generator: for every operationally correct program without user annotations, there exists a set of annotations that makes the verification condition generator output a valid formula.

Certified Implementation. We have proved in Coq that our translation is sound and we have implemented this translation in DVerify and tested it to be sound on a considerable number of samples. However, there is no proof that our implementation in F# is sound or indeed implements our proven translation. Coq has the ability to extract OCaml code from Coq source files [5]. This feature could be used to create a certified implementation by extracting our Coq translation function as OCaml code and using it as part of our F# project. To make this extracted code produce proper Boogie programs in practise, we would have to deal with a large number of implementation details we have ignored so far. For example, we would need to deal with the shallow embedding of the logic in Coq and relate our formalisation of Bemol to real Boogie.

Acknowledgements We thank Andrew D. Gordon for his helpful comments as well as the BOOGIE 2011 and CPP 2011 reviewers for their very useful feedback. Microsoft Research made our work much easier by making the Dminor source code and test suite available to us. Cătălin Hriţcu was supported by a fellowship from Microsoft Research and the International Max Planck Research School for Computer Science.

References

1. Bytecode level specification language and program logic. Mobius Project, Deliverable D3.1, 2006.
2. The Microsoft code name "M" modeling language specification, October 2009. `http://msdn.microsoft.com/en-us/library/dd548667.aspx`.
3. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *4th International Symposium on Formal Methods for Components and Objects (FMCO)*, Lecture Notes in Computer Science, pages 364–387. Springer, 2005.
4. M. Barnett, K. Leino, and W. Schulte. The Spec# programming system: An overview. In *Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, pages 49–69. Springer, 2005.
5. B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J. Filliâtre, E. Giménez, H. Herbelin, et al. The Coq proof assistant reference manual, version 8.2. INRIA, 2009.
6. J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems*, 33(2):8, 2011.
7. G. M. Bierman, A. D. Gordon, C. Hriţcu, and D. Langworthy. Semantic subtyping with an SMT solver. In *15th ACM SIGPLAN International Conference on Functional programming (ICFP 2010)*, pages 105–116. ACM Press, 2010.

8. S. Böhme, K. R. M. Leino, and B. Wolff. HOL-Boogie - an interactive prover for the Boogie program-verifier. In *21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 150–166. Springer, 2008.
9. E. Cohen, M. Moskal, S. Tobies, and W. Schulte. A precise yet efficient memory model for C. *Electronic Notes in Theoretical Computer Science*, 254:85–103, 2009.
10. M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC: Contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering (ICSE)*, pages 429–430. IEEE, 2009.
11. R. DeLine and K. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.
12. D. Detlefs, G. Nelson, and J. Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM (JACM)*, 52(3):473, 2005.
13. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *19th International Conference on Computer Aided Verification (CAV)*, pages 173–177. Springer, 2007.
14. H. Hosoya and B. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.
15. R. Jhala, R. Majumdar, and A. Rybalchenko. HMC: Verifying functional programs using abstract interpreters. Accepted at CAV, 2011. To appear.
16. T. Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11(5):541–566, 1999.
17. K. Knowles, A. Tomb, J. Gronski, S. Freund, and C. Flanagan. SAGE: Unified hybrid checking for first-class types, general refinement types and `Dynamic`. Technical report, UCSC, 2007.
18. N. Kobayashi and C.-H. L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *24th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 179–188. IEEE Computer Society, 2009.
19. H. Lehner and P. Müller. Formal translation of bytecode into BoogiePL. *Electronic Notes in Theoretical Computer Science*, 190(1):35–50, 2007.
20. K. R. M. Leino. This is Boogie 2. TechReport, 2008.
21. K. R. M. Leino, T. Millstein, and J. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 55(1-3):209–226, 2005.
22. C. Marinos. An introduction to functional programming for .NET developers. *MSDN Magazine*, April 2010.
23. E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, page 706. ACM, 2006.
24. J. Morris. Comments on "procedures and parameters". Undated and unpublished.
25. M. Naik and J. Palsberg. A type system equivalent to a model checker. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(5):29, 2008.
26. T. Nipkow. Hoare logics in Isabelle/HOL. In *Proof and System-Reliability*, pages 341–367. Kluwer, 2002.
27. B. Pierce, C. Casinghino, M. Greenberg, V. Sjöberg, and B. Yorgey. *Software Foundations*. `http://www.cis.upenn.edu/~bcpierce/sf/`, 2010.
28. S. Ranise and C. Tinelli. The satisfiability modulo theories library (SMT-LIB). `www.SMT-LIB.org`, 2006.
29. P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI)*, pages 159–169, 2008.
30. N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In *Proc. 19th European Symposium on Programming (ESOP 2010)*, pages 529–549, 2010.
31. T. Tarrach. Automatically verifying "M" modeling language constraints. Master's thesis, Saarland University, 2010.