GNATprove – a SPARK2014 verifying compiler

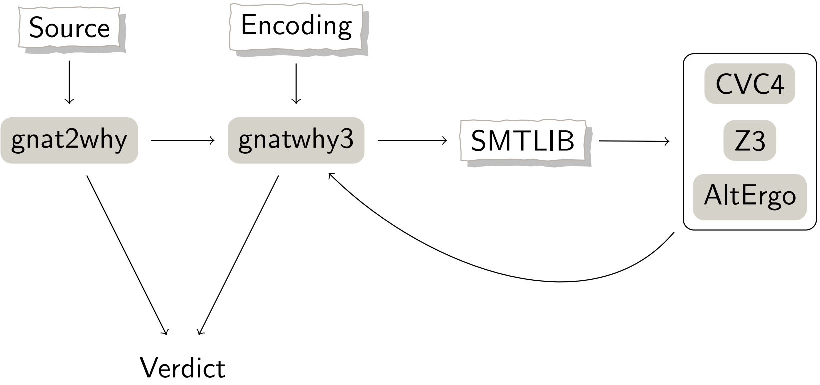
Florian Schanda, Altran UK

Tool architecture

User view

Tool architecture

More detailed view...



GNAT Frontend
Overview

- Ada 2012 and SPARK2014 lexer,
- parser,
- semantic analyser,
- expander,
- code generator (with gcc via intermediate language)

gnat2why

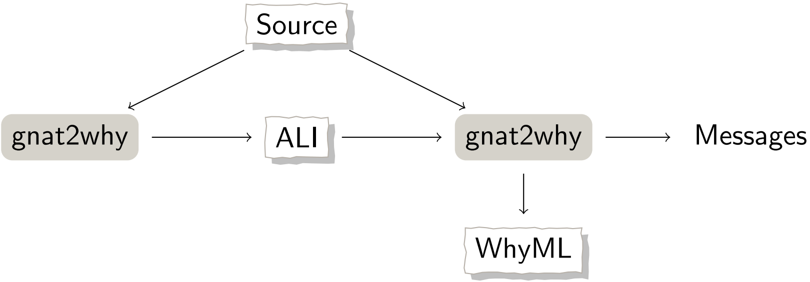
Overview

- Just another GNAT back-end
- An elaborate semantic analysis pass over the AST:

filter	Note which areas of the program are "in SPARK"
globals	Generate frame conditions (global contracts if they have not been specified) at varying levels of details
flow	Check initialization, non-aliasing, global contracts, and information flow contracts
translation	Transform SPARK subprograms into WhyML subprograms

gnat2why

Overview



gnat2why

Translation to WhyML

- SPARK is still an extremely complicated language
- Key properties need to be proven for a program to be correct ("verification conditions", or "VCs")
- Translation to a smaller, intermediate language WhyML
  - Simpler control flow
  - Simpler types
- Verification condition generation based on this IL

gnat2why

Translation to WhyML

```
function Example
   (A, B : Natural)
    return Natural
is
   R : Natural;
begin
   if A < B then
      R := A + 1;
   else
      R := B - 1;
   end if;
   return R;
end Example;
```

gnat2why

Translation to WhyML

```
function Example
   (A, B : Natural)
    return Natural
is
   R : Natural;
begin
   if A < B then
      R := A + 1;
   else
      R := B - 1;
   end if;
   return R;
end Example;
```

→

```
let example (a: int) (b: int)
    requires { a >= 0 /\ a <= 2147483647 }
    requires { b >= 0 /\ b <= 2147483647 }
    returns { r -> r >= 0 /\
                   r <= 2147483647 }
  = let r = ref 0 in
    if a < b then
       r := a + 1
    else
       r := b - 1;
    (!r)
```

gnat2why

Translation to WhyML

- Another traversal over AST (for SPARK), building another AST (for Why3)
- Tree is "pretty" printed, but not meant to be human readable
- One or more Why3 modules per SPARK entity
  - Types
  - Entity definitions, axioms
  - Subprogram definitions, axioms, bodies

  All of which are dumped into a single file for `gnatwhy3`.
- Not as nice as the previous example, a lot of extra information embedded:
  - Original source locations of all VCs
  - Checks ($x \neq 0$, or $x < 2^{32}$, etc.)

gnat2why

Translation to WhyML

Yep, not very readable... VC fragment for $r = a/b$:

```
( ( "GP_Sloc:overflow.adb:7:7" ( #"overflow.adb" 7 0 0#
overflow__example__result.int__content <- ( (
#"overflow.adb" 7 0 0# "GP_Sloc:overflow.adb:7:16"
"GP_Shape:return__div" "keep_on_simp" "model_vc"
"GP_Reason:VC_OVERFLOW_CHECK" "GP_Id:1"
(Standard__integer.range_check_(( #"overflow.adb" 7 0 0#
"GP_Reason:VC_DIVISION_CHECK" "GP_Id:0"
"GP_Sloc:overflow.adb:7:16" "GP_Shape:return__div"
"keep_on_simp" "model_vc" (Int_Division.div_
(Overflow__example__a.a) (Overflow__example__b.b))
))) ) ); #"overflow.adb" 7 0 0# raise Return__exc ) );
#"overflow.adb" 3 0 0# raise Return__exc )
```

gnat2why

Translation to WhyML

Yep, not very readable... VC fragment for $r = a/b$:

```
( ( "GP_Sloc:overflow.adb:7:7" ( #"overflow.adb" 7 0 0#
overflow__example__result.int__content <- ( (
#"overflow.adb" 7 0 0# "GP_Sloc:overflow.adb:7:16"
"GP_Shape:return__div" "keep_on_simp" "model_vc"
"GP_Reason:VC_OVERFLOW_CHECK" "GP_Id:1"
(Standard__integer.range_check_(( #"overflow.adb" 7 0 0#
"GP_Reason:VC_DIVISION_CHECK" "GP_Id:0"
"GP_Sloc:overflow.adb:7:16" "GP_Shape:return__div"
"keep_on_simp" "model_vc" (Int_Division.div_
(Overflow__example__a.a) (Overflow__example__b.b))
))) ) ); #"overflow.adb" 7 0 0# raise Return__exc ) );
#"overflow.adb" 3 0 0# raise Return__exc )
```

But we eventually get nice output...

overflow.adb:7:16: medium: divide by zero might fail (e.g. when B = 0)
overflow.adb:7:16: medium: overflow check might fail

gnat2why

Translation to WhyML

Features of the IL:

- Based on first order logic + theories
- In vague ML syntax with programming constructs:
  - (mutable) variables
  - sequences
  - loops, if, etc.
  - assertions
  - exceptions
- Built-in types are Boolean, Int, Real, Arrays, Records, Lists, Sets, etc. but more can be defined

gnat2why

Translation to WhyML

All checks come from a specification:

- Some checks are user defined (user asserts, postconditions)
- Ada RM defines basic checks (overflow, range, index, division by zero, discriminants, etc.)
- SPARK RM defines more (LSP checks, loop variants and invariants, etc.)

... we just follow that spec, and err on side of redundant checks.

SAT, SMT and SMTLIB

Recap: we now have the SPARK program in a different language (WhyML), but have not verified much...

- It's still difficult to prove anything, so we need to start talking to (automatic) theorem provers
- Language of choice is SMTLIB, but others exist
- So, next step is another language transformation

SAT, SMT and SMTLIB

Theories

Many theories have been implemented:

- Boolean
- Integer
- Reals
- Quantifiers
- Arrays
- Uninterpreted functions
- Bitvectors
- IEEE-754 Floating Point
- Strings
- Sets
- Algebraic Datatypes

SAT, SMT and SMTLIB

Overview of SMTLIB

- In the beginning all SMT solvers used their own input language
- This made it hard to compare solvers
- SMTLIB is both a standard language and a huge library of benchmarks
- SMTLIB only describes a search problem
- No control flow (if statements, loops, etc.) - so very far away from "programming language"

SAT, SMT and SMTLIB

SMTLIB is just s-expressions – I hope you remember your LISP?

```
; quantifier-free linear integer arithmetic
(set-logic QF_LIA)
; declarations
(declare-const x Int)
(declare-const y Int)
; hypothesis - things we know are true
(assert (<= 1 x 10)) ; 1 ≤ x ≤ 10
(assert (<= 1 y 10)) ; 1 ≤ y ≤ 10
; goal - what we want to prove
(define-const goal Bool (< (+ x y) 15)) ; x + y < 15
; search for a model where the goal is not true
(assert (not goal))
(check-sat)
```

SAT, SMT and SMTLIB

SMTLIB is just s-expressions – I hope you remember your LISP?

```
; quantifier-free linear integer arithmetic
(set-logic QF_LIA)
; declarations
(declare-const x Int)
(declare-const y Int)
; hypothesis - things we know are true
(assert (<= 1 x 10)) ; 1 ≤ x ≤ 10
(assert (<= 1 y 10)) ; 1 ≤ y ≤ 10
; goal - what we want to prove
(define-const goal Bool (< (+ x y) 15)) ; x + y < 15
; search for a model where the goal is not true
(assert (not goal))
(check-sat)
```

CVC4 output

```
sat
((x 10) (y 5))
```

SAT, SMT and SMTLIB

SMTLIB language overview

- Functions

```
(define-fun double (Int) Int)
(declare-fun triple ((x Int)) Int (+ x x x))
```

- Assertions and function calls

```
(assert (forall ((x Int)) (= (double x) (+ x x))))
```

- Predefined functions for theories

Core	=, =>, and, or, xor, not, ite, ...
Ints	+, -, *, /, >, >=, ...
Arrays	select, store
BV	bvadd, bvudiv, bvsdiv, bvlte, ...
FP	fp.add, fp.mul, fp.eq, fp.isInfinite, ...

SAT, SMT and SMTLIB

You can encode difficult problems with this...

```
(declare-fun fib (Int) Int)

(assert (= (fib 0) 0))
(assert (= (fib 1) 1))

; read this as: ∀x ∈ Int • x ≥ 2 ⟹ fib(x) = fib(x − 2) + fib(x − 1)
(assert (forall ((x Int))
          (=> (>= x 2)
              (= (fib x) (+ (fib (- x 2))
                            (fib (- x 1)))))))

; let's try to prove fib(10) < 10
(assert (not (< (fib 10) 10)))
(check-sat)
```

SAT, SMT and SMTLIB

You can encode difficult problems with this...

```
(declare-fun fib (Int) Int)

(assert (= (fib 0) 0))
(assert (= (fib 1) 1))

; read this as: ∀x ∈ Int • x ≥ 2 ⟹ fib(x) = fib(x − 2) + fib(x − 1)
(assert (forall ((x Int))
          (=> (>= x 2)
              (= (fib x) (+ (fib (- x 2))
                             (fib (- x 1)))))))

; let's try to prove fib(10) < 10
(assert (not (< (fib 10) 10)))
(check-sat)
```

CVC4 output

```
unknown
(((fib 10) 55))
```

SAT, SMT and SMTLIB

Solvers

Many solvers exist - (partial) table from Wikipedia:

Platform			Features						Notes
Name	OS	License	SMT-LIB	CVC	DIMACS	Built-in theories	API	SMT-COMP [2]	
ABsolver	Linux	CPL	v1.2	No	Yes	linear arithmetic, non-linear arithmetic	C++	no	DPLL-based
Alt-Ergo	Linux, Mac OS, Windows	CeCILL-C (roughly equivalent to LGPL)	partial v1.2 and v2.0	No	No	empty theory, linear integer and rational arithmetic, non-linear arithmetic, polymorphic arrays, enumerated datatypes, AC symbols, bitvectors, record datatypes, quantifiers	OCaml	2008	Polymorphic first-order input language à la ML, SAT-solver based, combines Shostak-like and Nelson-Oppen like approaches for reasoning modulo theories
Barcelogic	Linux	Proprietary	v1.2			empty theory, difference logic	C++	2009	DPLL-based, congruence closure
Beaver	Linux, Windows	BSD	v1.2	No	No	bitvectors	OCaml	2009	SAT-solver based
Boolector	Linux	GPLv3	v1.2	No	No	bitvectors, arrays	C	2009	SAT-solver based
CVC3	Linux	BSD	v1.2	Yes		empty theory, linear arithmetic, arrays, tuples, types, records, bitvectors, quantifiers	C/C++	2010	proof output to HOL
CVC4	Linux, Mac OS, Windows	BSD	Yes	Yes		rational and integer linear arithmetic, arrays, tuples, records, inductive data types, bit-vectors, strings, and equality over uninterpreted function symbols	C++	2010	version 1.4 released July 2014
Decision Procedure Toolkit (DPT)	Linux	Apache	No				OCaml	no	DPLL-based
iSAT	Linux	Proprietary	No			non-linear arithmetic		no	DPLL-based
MathSAT	Linux	Proprietary	Yes		Yes	empty theory, linear arithmetic, bitvectors, arrays	C/C++, Python, Java	2010	DPLL-based
MiniSmt	Linux	LGPL	partial v2.0			non-linear arithmetic		2010	SAT-solver based, Yices-based
OpenCog	Linux	AGPL	No	No	No	probabilistic logic, arithmetic. relational models	C++, Scheme, Python	no	subgraph isomorphism
OpenSMT	Linux, Mac OS, Windows	GPLv3	partial v2.0		Yes	empty theory, differences, linear arithmetic, bitvectors	C++	2011	lazy SMT Solver
SatEEn	?	Proprietary	v1.2			linear arithmetic, difference logic	none	2009	
SMTInterpol	Linux, Mac OS, Windows	LGPLv3	v2.0			uninterpreted functions, linear real arithmetic, and linear integer arithmetic	Java	2012	Focuses on generating high quality, compact interpolants.
SMCHR	Linux, Mac OS, Windows	GPLv3	No	No	No	linear arithmetic, nonlinear arithmetic, heaps	C	no	Can implement new theories using Constraint Handling Rules.
SMT-RAT	Linux, Mac OS	MIT	v2.0	No	No	linear arithmetic, nonlinear arithmetic	C++	2015	Toolbox for strategic and parallel SMT solving consisting of a collection of SMT compliant implementations.
SONOLAR	Linux, Windows	Proprietary	partial v2.0			bitvectors	C	2010	SAT-solver based
Spear	Linux, Mac OS, Windows	Proprietary	v1.2			bitvectors		2008	
STP	Linux, OpenBSD, Windows, Mac OS	MIT	partial v2.0	Yes	No	bitvectors, arrays	C, C++, Python, OCaml, Java	2011	SAT-solver based
SWORD	Linux	Proprietary	v1.2			bitvectors		2009	
UCLID	Linux	BSD	No	No	No	empty theory, linear arithmetic, bitvectors, and constrained lambda (arrays, memories, cache, etc.)		no	SAT-solver based, written in Moscow ML. Input language is SMV model checker. Well-documented!

... different strengths and logic support.

Why3 and WP

- So - Spark/WhyML and SMTLIB are quite different
- Last step is to go from the intermediate language to verification conditions expressed in SMTLIB