Verification of a Result Checker for Priority Queues

Hans de Nivelle

(joint work with Ruzica Piskac)

Nancy, 6 April 2006

Introduction

- Verification is an important topic. We need to develop techniques for theorem prover assistance, calculi for representation of large correctness proofs. (so that verification will become more and more economical)
- Our initial goal was to obtain insight in the requirements for theorem proving software verification.
- We did not exactly verify an implementation (of priority queues) but a result checker for priority queues. The reason that we did this is accidential. (The algorithms and complexity group asked us to do this)

Result Checking for Programs

Let F be a program that implements some function f. A result checker for F is an additional program P that computes the function p defined by

$$p(x,y) = \textbf{true} \text{ if } y = f(x),$$
$$p(x,y) = \textbf{false} \text{ if } y \neq f(x).$$

Program P should be not too costly. In addition, its correctness should be 'evident'.

Let f be some function. We call the pair of programs (W,P) result checker with witnesses for F if

- $w(x)$ is defined whenever $f(x)$ is,
- $p(x,y,w(x)) = \textbf{true}$ if $y = f(x)$,
- If $y \neq f(x)$, then for all z, $\quad p(x,y,z) = \textbf{false}$.

In order to be useful, W should be not too costly. Also P should be not expensive. In addition, its correctness must be ’evident’.

For many algorithms, this can be achieved.

In LEDA, the long term goal is to develope result checkers for all algorithms.

Result Checking for Data Structures

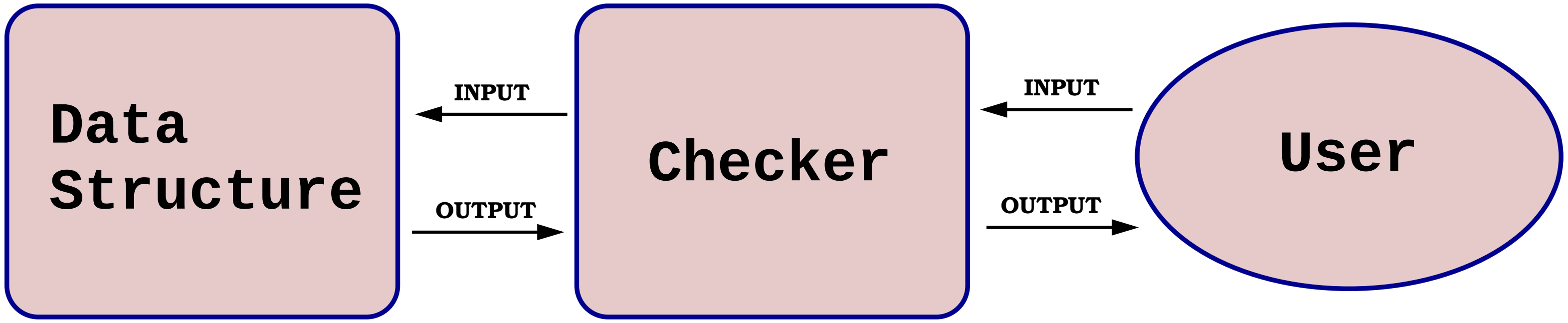
Datastructures normally exist for unbounded time.

During this time, they accept many inputs, and return many values.

Simple result checking would mean: Checking each result against all input before it. That is not feasible.

Only result checking with a witness is feasible.

The witness hast to be another datastructure that is updated in parallel with the main datastructure.



Priority Queues

A priority queue is a container which allows efficient retrieval of minimal elements under some quasi order $\leq$.

Definition:

A quasi-order over D is a relation $\leq$, which is

- transitive: $\forall d_1, d_2, d_3, \quad d_1 \leq d_2 \wedge d_2 \leq d_3 \rightarrow d_1 \leq d_3$.
- total: $\forall d_1, d_2, \quad d_1 \leq d_2$ or $d_2 \leq d_1$.

Given a quasi-ordered set $(D, \leq)$, an element $\perp \in D$ is called bottom element if $\forall d \in D, \ \perp \leq d$. We call the pair $(D, \leq)$ a QOS, and the triple $(D, \leq, \perp)$ a QOSB.

Priority Queue (2)

Definition:

Let $(D, \leq)$ be QOS. A priority queue over $(D, \leq)$ is a container supporting the following operations:

- create_{PQ}.
- $\mathsf{insert}(d)$.
- $\mathsf{remove}(d)$.
- find_min.
- remove_min.

Note that

- find_min, remove_min are incompletely specified.
- We are slightly cheating with remove.

Checking Priority Queues

How to check a priority queue?

Obvious Solution: Whenever find_min, remove_min returns element, go through the priority queue, and check that there is no smaller element.

Problem: This is too costly. A PQ finds minimal element in time $O(\log(n))$. Going through all elements takes time $O(n)$.

Solution: Postpone detection of errors. An error is detected, not when a non-minimal element is returned, but at the moment when a smaller element is returned.

Such a checker is called off-line.

THEOREM: (Mehlhorn+ Finkler, 1999) There is no on-line, time superefficient checker for PQs.

Checking Priority Queues (2)

An efficient checker can be obtained using lower bounds.

Suppose that the priority queue contains two elements 1 and 2 and that find_min returns 2.

Then the priority queue implicitly claims that 2 is smaller than 1.

One can decorate 1 with the set of elements that the priority queue considered smaller. If there is an element in this set that is not smaller, then there is a problem.

Since only the maximum of the elements in the set counts, it is enough to attach to each element in the priority queue, the maximum of the elements returned by find_min during the time that the element was in the queue. We call this element the lower bound.

We call the data structure that attaches to each element in the priority queue a lower bound, the system of lower bounds.

Checking Priority Queues (3)

Operation	SLB after the operation
$pq.\mathsf{insert}(1)$	$(\ (1,\perp)\ )$
$pq.\mathsf{insert}(2)$	$(\ (1,\perp),\ (2,\perp)\ )$
$pq.\mathsf{insert}(3)$	$(\ (1,\perp),\ (2,\perp),(3,\perp)\ )$
$pq.\mathsf{find\_min} \Rightarrow 2$	$(\ (1,2),\ (2,2),\ (3,2)\ )$
$pq.\mathsf{remove\_min} \Rightarrow 2$	$(\ (1,2),\ (3,2)\ )$
$pq.\mathsf{remove}(3)$	$(\ (1,2)\ )$
$pq.\mathsf{remove}(1)$	$\Rightarrow$ lower bound violated.

What if offended element is never retrieved? Do a periodic check. Cheap if one does it rarely enough.

A checker using SLBs can be implemented in O(1) (amortized) time.

Formal Framework

We want to prove that whenever the system of lowerbounds accepts some object, then this object has behaved like a priority queue at least until it was checked.

Wanted: A framework that can express things like:

- Implementation I implements some data structure D.
- Implementation I has behaved like a D up to some moment t.
- Implementation I has behaved like a D on some set of inputs.

Some Notation

In object-oriented languages, the notation $pq.\mathsf{remove\_min}$ is used to indicate that $\mathsf{remove\_min}$ is a method of pq.

The method $\mathsf{remove\_min}$ defines two functions at the same time: One returns the minimal element, the other deletes it from the priority queue. We keep the notation $pq.\mathsf{remove\_min}$ for the first function, and introduce the notation $pq/\mathsf{remove\_min}$ for the second.

Using this notation, one can write:

$$pq.\mathsf{remove\_min} = pq.\mathsf{find\_min},$$

$$pq/\mathsf{remove\_min} = pq/\mathsf{remove}(pq.\mathsf{find\_min}).$$

Implementation: Attempt 1

1. For each method m of $P^{\#}$, assume that its signature has form $m(\overline{x},\overline{y})$, where $\overline{x}$ are the parameters of type $P^{\#}$ and $\overline{y}$ are the parameters of other types.
2. Furthermore, assume that each $p.m(\overline{x},\overline{y})$ is not of type $P^{\#}$, and that each $p/m(\overline{x},\overline{y})$ is of type $P^{\#}$.

A type P implements a type $P^{\#}$ with implementation function $I : P \rightarrow P^{\#}$ if for every method $m^{\#}(\overline{x},\overline{y})$ of $P^{\#}$, there exists a corresponding method $m(\overline{x},\overline{y})$ of P, and for each method $m^{\#}(\overline{x},\overline{y})$ of $P^{\#}$:

1. $p.m(\overline{x},\overline{y}) = I(p).m^{\#}(I(\overline{x}),\overline{y})$.
2. $I(\ p/m(\overline{x},\overline{y})\ ) = I(p)/m^{\#}(I(\overline{x}),\overline{y})$.
3. $I(\ \mathsf{create}_P\ ) = \mathsf{create}_{P^{\#}}$.

Implementation: Attempt 2

What is missing? $\Rightarrow$ non-determinism and partiality.

In order to deal with non-determinism in $P^{\#}$, we add extra parameters to the methods of $P^{\#}$, which P has to fill in. Write $m_{\overline{z}}(\overline{x},\overline{y})$ for a method of $P^{\#}$ with implicit parameters $\overline{z}$. For each method $m_{\overline{z}}(\overline{x},\overline{y})$ of $P^{\#}$, there has to be a corresponding method $m(\overline{x},\overline{y})$ of P, s.t.

1. $\forall p \in P \ \ \forall \overline{x} \in P \ \ \forall \overline{y} \ \ \exists \overline{z} \ \ p.m(\overline{x},\overline{y}) = I(p).m_{\overline{z}}(I(\overline{x}),\overline{y})$.
2. $\forall p \in P \ \ \forall \overline{x} \in P \ \ \forall \overline{y} \ \ \exists \overline{z} \ \ I(\ p/m(\overline{x},\overline{y})\ ) = I(p/m_{\overline{z}}(I(\overline{x}),\overline{y})$.
3. $I(\ \mathsf{create}_P\ ) = \mathsf{create}_{P^{\#}}$.

Now brace for partiality ...

Implementation: Attempt 3

Assume that method $p.m_{\overline{z}}(\overline{x},\overline{y})$ has precondition $\Pi^{\#}(p,\overline{x},\overline{y},\overline{z})$ on $P^{\#}$.
Define $\Pi(p,\overline{x},\overline{y}) := \exists\overline{z}\ \ P^{\#}(I(p),I(\overline{x}),\overline{y},\overline{z})$.

For each method $p.m_{\overline{z}}(\overline{x},\overline{y})$ of $P^{\#}$ with precondition $\Pi^{\#}(p,\overline{x},\overline{y},\overline{z})$, there is a method $p.m(\overline{x},\overline{y})$ of P with precondition $\Pi(p,\overline{x},\overline{y})$, s.t.

1. $\forall p \in P\ \ \forall\overline{x} \in P\ \ \forall\overline{y}\ \ \Pi(p,\overline{x},\overline{y}) \rightarrow$
   $$\exists\overline{z}\ \ \Pi^{\#}(I(p),I(\overline{x}),\overline{y},\overline{z})\ \wedge\ p.m(\overline{x},\overline{y}) = I(p).m_{\overline{z}}(I(\overline{x}),\overline{y}).$$

2. $\forall p \in P\ \ \forall\overline{x} \in P\ \ \forall\overline{y}\ \ \Pi(p,\overline{x},\overline{y}) \rightarrow$
   $$\exists\overline{z}\ \ \Pi^{\#}(I(p),I(\overline{x}),\overline{y},\overline{z})\ \wedge\ I(\ p/m(\overline{x},\overline{y})\ ) = I(p)/m_{\overline{z}}(I(\overline{x}),\overline{y}).$$

3. $I(\ \mathsf{create}_P\ ) = \mathsf{create}_{P^{\#}}.$

Axiomatization of Priority Queues (1)

Method contains and insert have no preconditions:

$$\text{create.contains}(p) = \text{false},$$
$$pq/\text{insert}(p).\text{contains}(p) = \text{true},$$
$$pq/\text{insert}(p_1).\text{contains}(p_2) = pq.\text{contains}(p_1) \text{ if } p_1 \neq p_2.$$

Method empty has no preconditions:

$$\text{create}_{PQ}.\text{empty} = \text{true},$$
$$pq/\text{insert}(p).\text{empty} = \text{false}.$$

Method remove(p) has precondition $pq.\text{contains}(p)$:

$$pq/\text{insert}(p)/\text{remove}(p) = pq.$$
$$pq/\text{insert}(p_1)/\text{remove}(p_2) = pq/\text{remove}(p_2)/\text{insert}(p_1)$$
$$\text{if } p_1 \neq p_2.$$

Axiomatization of Priority Queues (2)

Methods find_min and remove_min are non-deterministic. They have an implicit parameter p that specifies which minimal element will be returned.

Method $\mathsf{find\_min}_p$ and $\mathsf{remove\_min}_p$ have precondition:

$$pq.\mathsf{contains}(p) \wedge \forall p' \; pq.\mathsf{contains}(p') \rightarrow p \leq p'.$$

The axioms are:

$pq.\mathsf{find\_min}_p = p.$

$pq.\mathsf{remove\_min}_p = p.$

$pq/\mathsf{remove\_min}_p = pq/\mathsf{remove}(p).$

One can also add extensionality:

$pq/\mathsf{insert}(p_1)/\mathsf{insert}(p_2) = pq/\mathsf{insert}(p_2)/\mathsf{insert}(p_1).$

Behaviour of implementation on find_min (1)

We show that the notion of implementation does what it is supposed to do for the method find_min :

On the abstract level, $pq^{\#}.\mathsf{find\_min}_p$ has precondition

$$\Pi^{\#}(pq^{\#}, p) := pq^{\#}.\mathsf{contains}(p) \wedge \forall p' \; pq^{\#}.\mathsf{contains}(p') \rightarrow p \leq p'.$$

On the concrete level, pq must have a method find_min with precondition

$$\Pi(pq) := \exists p \;\; I(pq).\mathsf{contains}(p) \wedge \forall p' \; I(pq).\mathsf{contains}(p') \rightarrow p \leq p'.$$

Behaviour of implementation on find_min (2)

$$\Pi(pq) := \exists p \;\; I(pq).\mathsf{contains}(p) \wedge \forall p' \; I(pq).\mathsf{contains}(p') \rightarrow p \leq p'.$$

Since $I(pq).\mathsf{contains}(p) = pq.\mathsf{contains}(p)$, this is equivalent to

$$\Pi(pq) \leftrightarrow \exists p \;\; pq.\mathsf{contains}(p) \wedge \forall p' \; pq.\mathsf{contains}(p') \rightarrow p \leq p'.$$

It can be shown by induction that this is equivalent to

$$\Pi(pq) \leftrightarrow pq.\mathsf{empty} = \mathsf{false}.$$

This is the expected precondition for find_min.

Behaviour of implementation on find_min (3)

For each (concrete) priority queue pq, condition 1 equals

$$\Pi(pq) \rightarrow \exists p \;\; \Pi^{\#}(pq, p) \wedge pq.\mathsf{find\_min} = I(pq).\mathsf{find\_min}_p.$$

Because $I(pq).\mathsf{find\_min}_p = p$, this can be simplified into

$$\Pi(pq) \rightarrow \exists p \;\; \Pi^{\#}(pq, p) \wedge pq.\mathsf{find\_min} = p.$$

This in turn is equivalent to

$$\Pi(pq) \rightarrow \Pi^{\#}(pq, pq.\mathsf{find\_min}\ ).$$

This is the usual axiom for find_min.

Checked Priority Queue (1)

Definition: A priority queue pretender is an object that pretends to be a priority queue.

A checked priority queue is an ordered triple (pqp, s, ok), where pqp is a priority queue pretender, s is a system of lower bounds (essentially a list of ordered pairs), and ok is a boolean.
Some characteristic axioms:

$$\mathsf{create}_{CPQ} = (\mathsf{create}_{PQP}, \mathsf{create}_{LBS}, \mathsf{true}).$$

$$(pqp, s, ok)/\mathsf{insert}(p) = (pqp/\mathsf{insert}(p), s/\mathsf{assign}(p, \perp), ok).$$

Checked Priority Queue (2)

If s.empty $=$ true, then

$(pqp, s, ok).\mathsf{find\_min} = \perp$, and

$(pqp, s, ok)/\mathsf{find\_min} = (pqp, s, \mathsf{false})$.

Otherwise, put $p := pqp.\mathsf{find\_min}$. Then

$(pqp, s, ok).\mathsf{find\_min} = p$.

If $s.\mathsf{contains}(p)$ and $s.\mathsf{lowerbound}(p) \leq p$, then

$$(pqp, s, ok)/\mathsf{find\_min} = (pqp/\mathsf{find\_min}, s/\mathsf{update}(p), ok),$$

otherwise

$$(pqp, s, ok)/\mathsf{find\_min} = (pqp/\mathsf{find\_min}, s, \mathsf{false}).$$

The other methods have similar specifications.

Testing, Result Checking

This slide should have contained Attempt 4, but compassion with the audience prevented me from writing it down.

It seems we forgot that we are checking and not verifying. $\Rightarrow$ Tests and result checks are partial verifications.

Definition We say that a type P conditionally implements $P^{\#}$ (with some condition ϕ) if conditions 1,2 hold, when relativized to ϕ,

$$\forall p \in P \ \ \phi(p) \rightarrow \cdots$$

we proved exactly

Definition: We define the following relation $\prec$ on checked priority queues: $cpq_1 \prec cpq_2$ if there is a sequence of operations $M_1, \ldots, M_n$, s.t.

$$cpq_1/M_1/\cdots/M_n = cpq_2.$$

Theorem(formally verified)
Define

$$\phi(cpq') := \exists cpq_F \;\; cpq' \leq cpq_F \wedge cpq_F.ok \wedge cpq_F.check.$$

The concrete data type CPQ conditionally implements the abstract datatype PQ on condition ϕ.

Some Lessons learnt for theorem proving

- Partial Functions/Subtyping are essential in the modelling of behaviour of programs.
- Theorem provers need better termination behaviour, and also more predictability. Predictability is probably more important than strength.

Lessons learnt for interactive verification

Interactive verifiers need a modelling mechanism which is similar to the class mechanism:

An interface consists of:

1. A domain. (possibly set of domains)
2. A set of functions and predicates over the domain.
3. A set of axioms specifying the behaviour of (2) on (1).
4. Procedural information. (which equalities/equivalences are rewrite rules, which Horn clauses are typing rules)

A class contains only 1,2,4. (Because properties are implicit in programming languages)

Use of interface mechanism

1. Prove properties of the domain (1) and its functions/predicate (2) using the axioms (3).

2. Prove that some set D and some functions/predicates satisfy the axioms (3).

   Then the procedures in (4) come for free, together with all properties that have been proven about the interface.

   (This is a higher-order mechanism)

Inheritance

In C^{++}, one can write:

class A : public B $\{\cdots\}$.

virtual void B :: print( ) $\{\cdots\}$;

In object-oriented languages, the programmer can choose for A to keep B :: print( ), or to define a new print( ).

Inheritance in the program disappears in the specification.

Results and Future Work

- We formally proved (using Saturate) the correctness of the checker for priority queues.
- We gave a functional framework in which it can be formally expressed what a result checker is. In the same framework, it can also be expressed what an implementation of a specification is.
- For theorem proving, we learnt that one needs richer type systems, and better termination.
- For interactive verification, we think that one needs a structuring mechanism based on interfaces.