universität innsbruck

Summer Term 2024

Program Verification

Part 7 – Certification

René Thiemann

Department of Computer Science

Certification

Certification – Motivation

- situation
  - program verification is work intensive
  - verification might be too expensive for complex programs
- work-around via certification
  - assume there is some complex function f implemented in some program
  - property $P(x, f(x))$ should be satisfied for all x
  - to this end implement a slightly extended function f_e such that
    - $f_e(x)$ computes the pair $(f(x), c(x))$ where $c(x)$ is a certificate for input x, and
    - certification is possible: given $x, f(x), c(x)$ one can check $P(x, f(x))$ with a simple program (certifier), and ideally, this simple program is completely verified
- advantages of certification
  - no need to verify the complex programs
  - one certifier can check the certificates of many similar complex programs (assumption: common certificate format)
- disadvantages of certification
  - certificates might be refused (incorrect answers of complex programs or incomplete certifier)
  - overhead in certificate generation and checking

Certification – Examples

- matrix-matrix multiplication: $f(A, B) = A \times B$
  - no certification possible, just computation
- matrix-inversion: $f(A) = A^{-1}$ (for invertible inputs A)
  - certification possible without extra information
  - given A and A^{-1} it suffices to check $AA^{-1} = I$
  - matrix multiplication is easier to verify than an algorithm for matrix inversion
- SAT solving: $f(\varphi) = (\exists \alpha. [\![\varphi]\!]_\alpha = \top)$ for CNFs φ
  - certification possible for positive answers: provide α
  - certification possible for negative answers: provide resolution proof
  - common format is (variant of) DRAT (used in SAT competitions)
  - several independent certifiers; some of them are verified
- Termination analysis: $f(R) = SN(\to_R)$
  - certification possible: provide applied techniques with parameters and extra information
  - common format is CPF (used in termination competitions)
  - one certifier: CeTA (developed in Innsbruck)

Reduction Pairs

- task of termination analysis tool: find reduction pair such that constraints are satisfied
- task of certifier: given reduction pair, check that constraints are oriented
- different complexity of both tasks
  - only tool: choose suitable class of reduction pairs
  - only certifier: verify reduction pair properties of each class
  - lexicographic path order (LPO)
    search parameters: NP-complete; checking constraints: P
  - Knuth-Bendix Order (KBO)
    search parameters: P (complex algorithm); checking constraints: P (trivial algorithm)
  - linear polynomial interpretations
    search parameters: undecidable; checking constraints: P

Certificates for Applying Reduction Pairs

- question of format of certificate for (iterated) reduction pair application
- obvious idea: just provide parameters of pairs
- example
  - consider termination problem with 5 dependency pairs (DP 1 – DP 5)
  - termination tool internally applies
    RP 1, a polynomial interpretation with certain parameters P 1, to remove DP 2 and DP 3,
    then RP 2, some KBO with certain parameters P 2, to remove DP 1 and DP 4,
    and finally RP 3, some other polynomial interpretation with parameters P 3 to remove DP 5
  - structure of certificate

    Poly(P 1); KBO(P 2); Poly(P 3)
- problem in case of rejected certificates (e.g., if tool uses tuned version of some RP)
  - certifier might replay this proof, but remains with DP 1 in the end
  - with above certificate structure, it is not possible to localize failure
- easy solution: add more information into certificate

  Poly(P 1, delete DP 2,3); KBO(P 2, delete DP 1,4); Poly(P 3, delete DP 5)

Usable Equations

- task of termination analysis tool: compute usable equations to setup constraints
- task of certifier: given usable equations, check that these have been computed correctly
- reminder: let $\mathcal{E}$ be equations of program, let $\mathcal{P}$ be a set of dependency pairs; define $\mathcal{U}(\mathcal{P}) = \bigcup_{s \to t \in \mathcal{P}} \mathcal{U}(t)$ where $\mathcal{U}(t)$ is defined inductively as

$$\frac{t \trianglerighteq u \quad \ell = r \in \mathcal{E} \quad root(u) = root(\ell)}{\ell = r \in \mathcal{U}(t)}$$

$$\frac{\ell' = r' \in \mathcal{U}(t) \quad \ell = r \in \mathcal{U}(r')}{\ell = r \in \mathcal{U}(t)}$$

- difficulties
  - computing usable equations is a fixpoint algorithm (add new usable equations until nothing more is detected)
  - verification of fixpoint algorithms is sometimes tricky
  - tools implement different versions of usable equations (mixture of various optimizations, e.g., inclusion of argument filters, etc.)

  $\longrightarrow$ there is not the definition of usable equations
  - solution: certifier allows over-approximation of those usable equations that have been verified

Computation of Usable Equation (Non-Optimized Version)

- $\mathcal{U}(\mathcal{P}) = \bigcup_{s \to t \in \mathcal{P}} \mathcal{U}(t)$ where $\mathcal{U}(t)$ is defined inductively as

$$\frac{t \trianglerighteq u \quad \ell = r \in \mathcal{E} \quad root(u) = root(\ell)}{\ell = r \in \mathcal{U}(t)}$$

$$\frac{\ell' = r' \in \mathcal{U}(t) \quad \ell = r \in \mathcal{U}(r')}{\ell = r \in \mathcal{U}(t)}$$

- inductive definition: $\mathcal{U}(t)$ is least set such that inference rules are satisfied
- soundness proof reveals: $\mathcal{U}(t)$ can be any set such that inference rules are satisfied
- certification
  - demand that $\mathcal{U}(\mathcal{P})$ is provided in certificate
  - certification: check that above inference rules are satisfied
  - much easier than computing $\mathcal{U}(\mathcal{P})$ in verified way

Soundness Proof for Certification: Being Closed under Usable Equations

- fix $\mathcal{U}$ and $\mathcal{E}$
- definition: t is closed under usable rules ($closed(t)$) if

$$\forall u.\ t \trianglerighteq u \longrightarrow \ell = r \in \mathcal{E} \longrightarrow root(u) = root(\ell) \longrightarrow \ell = r \in \mathcal{U}$$

- lemma: assume $\forall \ell = r \in \mathcal{U}.\ closed(r)$; then

$$(\forall x. NF(\sigma(x))) \longrightarrow closed(t) \longrightarrow t\sigma \xhookrightarrow{\mathsf{i}}{}^{*}_{\mathcal{E}}\ u \longrightarrow t\sigma \xhookrightarrow{\mathsf{i}}{}^{*}_{\mathcal{U}}\ u$$

  by induction on (number of steps, size of t)
- remark: conditions in lemma (being closed) are easy to check
- proof case 1: assume $t\sigma \xhookrightarrow{\mathsf{i}}{}^{*}_{\mathcal{E}}\ \ell\delta \xhookrightarrow{\mathsf{i}} r\delta \xhookrightarrow{\mathsf{i}}{}^{*}_{\mathcal{E}}\ u$ where $\ell\delta \xhookrightarrow{\mathsf{i}} r\delta$ is first root step
  - by assumptions $root(t\sigma) = root(t) = root(\ell)$, hence $\ell = r \in \mathcal{U}$ and thus $closed(r)$
  - via IH obtain $t\sigma \xhookrightarrow{\mathsf{i}}{}^{*}_{\mathcal{U}}\ \ell\delta$ and $r\delta \xhookrightarrow{\mathsf{i}}{}^{*}_{\mathcal{U}}\ u$
- proof case 2: assume $t\sigma = f(t_1\sigma, \ldots, t_n\sigma) \xhookrightarrow{\mathsf{i}}{}^{*}_{\mathcal{E}}\ f(u_1, \ldots, u_n) = u$ (only non-root steps)
  - by definition $closed(t_i)$ and IH yields $t_i\sigma \xhookrightarrow{\mathsf{i}}{}^{*}_{\mathcal{U}}\ u_i$ for all $1 \leq i \leq n$

Nontermination via Loops

- a loop is a reduction of form $t \hookrightarrow^+ D[t\delta]$
- whenever program admits a loop, then it is non-terminating

$$t \hookrightarrow^+ D[t\delta] \hookrightarrow^+ D[D[t\delta]\delta] \hookrightarrow^+ D[D[D[t\delta]\delta]\delta] \hookrightarrow^+ \dots$$

- certificate of non-termination: provide t, D, δ and $t = t_1 \hookrightarrow t_2 \hookrightarrow \dots \hookrightarrow t_n = D[t\delta]$
- certification needs to check that every step is correct: given t_i and t_{i+1}, ensure $t_i \hookrightarrow t_{i+1}$
- approach 1: verified algorithm to compute all successors of t_i
  - requires verified matching algorithm, etc.
- approach 2: certificate contains additional information
  - require for every step $\ell = r \in \mathcal{E}$, C and σ such that

$$t_i = C[\ell\sigma] \wedge t_{i+1} = C[r\sigma]$$

  - then only the latter needs to be checked by certifier
  - disadvantage: bulky certificates, more tedious to generate
- approach 3: unverified algorithm in certifier computes $\ell = r$, C, and σ for each $t_i \hookrightarrow t_{i+1}$

Partially Verified Programs

- approach 3 on previous slide contains interesting idea
- verified programs can use unverified sub-algorithms to generate auxiliary information to simplify checking task
- this approach is used in big verified programs
- example: verified C compiler (CompCert)
  - correctness of C compiler has been formally verified

    for every C program P, if compiler(P) = A (assembly-code), then P and A are equivalent
  - many sub-algorithms of C compiler are fully verified
  - some algorithms use unverified programs to compute information that is then certified
  - if any of these unverified programs delivers faulty information, then compilation just fails

Example: Call-Graph Analysis

- during compilation, call-graph needs to be computed
- compilation handles each block of mutually recursive functions separately
- blocks correspond to strongly connected components (SCCs) of call-graph
- instead of verifying SCC algorithm, design certificate approach
- w.l.o.g., we consider graphs G where every node has a self-loop
  (no distinction between SCC $\{n\}$ and a node n that is not on any SCC)
- over-approximation
  - certificate contains list of SCC in topologic order $C_1, C_2, \ldots$
  - check that all nodes are covered by some C_i
  - topologic order: whenever $i < j$ then there is no edge from C_i to C_j
  - remark: many SCC-algorithms actually compute SCCs in (reverse) topological order
  - easy to verify: whenever S is SCC, then $S \subseteq C_i$ for some i
    - SCCs are non-empty, so pick some $s \in S$ and obtain i such that $s \in C_i$
    - now pick some arbitrary $t \in S$, hence $(s,t) \in G^*$ and $(t,s) \in G^*$
    - then obtain j such that $t \in C_j$
    - by topological order, obtain $j \geq i$ from $(s,t) \in G^*$ and similarly $i \geq j$, so $j = i$
    - hence $t \in C_i$, and by arbitrary choice of t, $S \subseteq C_i$

SCC Certification

- potential certificate for under-approximation
  - for each C_i in certificate, provide a cyclic path that contains all nodes of C_i
  - easy to certify and obviously correct
  - lemma: whenever criterion is satisfied, then each C_i is strongly connected
  - format of certificate is not optimal, cf. proseminar
- for some properties, it is not required to check minimality of components

Example – Completion

- task of completion: convert set of equations $\mathcal{E}$ into program $\mathcal{R}$ such that
  - $\mathcal{R}$ is confluent and terminating
  - $s =_{\mathcal{E}} t$ iff $s{\downarrow_{\mathcal{R}}} = t{\downarrow_{\mathcal{R}}}$
- certificate contains
  - proof of confluence and termination of $\mathcal{R}$
  - proofs of $\ell =_{\mathcal{E}} r$ for each $\ell = r \in \mathcal{R}$
- the latter proofs are obtained via recording completion
  - new equations $s = t$ are produced by overlapping two known equations $s =_{\mathcal{E}'} u =_{\mathcal{E}'} t$ for intermediate set of equations $\mathcal{E}'$
  - memoize for each generated equation how it has been produced
  - final $\mathcal{R}$ is just a subset of set of all equations that have been generated
  - expand each $\mathcal{R}$ until original equations are used
- problem: size of expansion might grow exponentially

Example Completion Run

- $\mathcal{E}$ consists of

$$\mathsf{f}(\mathsf{s}(x), y) = \mathsf{f}(x, \mathsf{c}(y, y)) \tag{1}$$

$$\mathsf{f}(\mathsf{0}, y) = \mathsf{g}(y) \tag{2}$$

$$\mathsf{g}(\mathsf{e}) = \mathsf{t} \tag{3}$$

$$\mathsf{and}(\mathsf{g}(y), \mathsf{g}(y)) = \mathsf{g}(\mathsf{c}(y, y)) \tag{4}$$

$$\mathsf{and}(\mathsf{t}, \mathsf{t}) = \mathsf{t} \tag{5}$$

$$\mathsf{start} = \mathsf{f}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0}))), \mathsf{e}) \tag{6}$$

and we derive

$$\mathsf{g}(\mathsf{c}(\mathsf{e}, \mathsf{e})) \stackrel{4}{=} \mathsf{and}(\mathsf{g}(\mathsf{e}), \mathsf{g}(\mathsf{e})) \stackrel{3}{=} \mathsf{and}(\mathsf{t}, \mathsf{t}) \stackrel{5}{=} \mathsf{t} \tag{7}$$

$$\mathsf{g}(\mathsf{c}(\mathsf{c}(\mathsf{e}, \mathsf{e}), \mathsf{c}(\mathsf{e}, \mathsf{e}))) \stackrel{4}{=} \mathsf{and}(\mathsf{g}(\mathsf{c}(\mathsf{e}, \mathsf{e})), \mathsf{c}(\mathsf{e}, \mathsf{e})) \stackrel{7}{=} \mathsf{and}(\mathsf{t}, \mathsf{t}) \stackrel{5}{=} \mathsf{t} \tag{8}$$

$$\mathsf{g}(\mathsf{c}(\mathsf{c}(\mathsf{c}(\mathsf{e}, \mathsf{e}), \mathsf{c}(\mathsf{e}, \mathsf{e})), \mathsf{c}(\mathsf{c}(\mathsf{e}, \mathsf{e}), \mathsf{c}(\mathsf{e}, \mathsf{e})))) \stackrel{4}{=} \mathsf{and}(\dots, \dots) \stackrel{8}{=} \mathsf{and}(\mathsf{t}, \mathsf{t}) \stackrel{5}{=} \mathsf{t} \tag{9}$$

$$\mathsf{start} \stackrel{6}{=} \mathsf{f}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0}))), \mathsf{e}) \stackrel{1,2}{=} \mathsf{g}(\mathsf{c}(\dots), \mathsf{c}(\dots)) \stackrel{9}{=} \mathsf{t} \tag{10}$$

Improved Certification for Completion

- do not fully expand to original equations, but allow (and certify) intermediate equations

Current Bachelor Project

- design automation and certificate format for similar task: rewriting induction

Summary

- certification is often, but not always applicable
- design of certificate format is crucial
  - should contain enough information to simplify certification task
  - should be easy to generate for tools
- certification approach can be used within fully verified programs: invoke unverified programs to enrich/generate certificates on the fly, to avoid task of full verification of complex algorithm