Masterprojekt "Prooftree2Isabelle"

Christian Schilling

13. November 2012

EDIT 2: [Erinnerungsliste 1-2 und Zeitplan]

EDIT 3: [Stand und Änderung des Zeitplans]

EDIT 4: [Erinnerungsliste 3]

1 Vorüberlegungen

1.1 Beschreibung

In meinem Masterprojekt werde ich die Möglichkeit implementieren, einen von *SMTInterpol*[1] generierten Beweis mit *Isabelle*[2] automatisch zu überprüfen.

SMTInterpol ist ein Theorembeweiser für Computerprogramme, geschrieben in Java (welches folglich auch die Programmiersprache für meine Anwendung sein wird). Er beherrscht zur Zeit die Theorie der Gleichheit mit uninterpretierten Funktionen und die Theorie der linearen Arithmetik. Ein Unerfüllbarkeitsbeweis in diesen Theorien basiert auf Resolutionsregeln; der jeweils verwendete Resolutionsbeweis kann generiert werden (im Folgenden *Beweisbaum*). Ziel ist es nun, diesen Resolutionsbeweis als korrekt zu verifizieren.

Dafür wird der interaktive Theorembeweiser Isabelle benutzt. Ein Beweisbaum soll in ein gültiges Eingabeformat umgewandelt und dessen Korrektheit automatisch bewiesen werden (sofern er denn korrekt ist).

1.2 Vorgehen

Es wird im Laufe der nächsten Wochen Änderungen am Format von SMTInterpol geben, daher ist diese Aufteilung nicht dogmatisch zu verstehen.

(1) Zunächst werde ich mir das Ausgabeformat eines Beweisbaums ansehen und die möglichen Regeln und Besonderheiten kennenlernen. Wahrscheinlich werde ich mich zunächst auf eine Theorie beschränken (aber die andere im Hinterkopf behalten), da sie doch relativ unterschiedlich funktionieren. Lineare Arithmetik scheint mir persönlich einfacher zu sein, siehe Kapitel 1.3.

(2) Isabelle muss natürlich die zu beweisende Aussage selbst verstehen. Die Formel kann ich direkt vom Parser erhalten, sodass ich hier auf Javaseite kaum etwas zu tun habe. Der Isabelle-Teil wird wahrscheinlich länger dauern, weil ich mich zunächst einarbeiten muss.

(3) Danach benötige ich für alle Regeln im Beweisbaum die Pendants in Isabelle. Teilweise gibt es diese sicherlich, teilweise müssen diese auch hinzugefügt werden. Eine Möglichkeit hierfür

[1] http://ultimate.informatik.uni-freiburg.de/smtinterpol/
[2] http://isabelle.in.tum.de/

ist es, Metaregeln zu schreiben, die ich dann einfach benutzen kann. Dadurch bleibt der Java-Code einfacher. Im Moment spricht nichts gegen dieses Vorgehen. Alternativ könnte man auch die längeren Regeln direkt von Java generieren lassen. Vermutlich werde ich das in einer ersten Version auch tun, wenn die Metaregeln doch mehr Aufwand bedeuten.

(4) Ist das geschafft, bleibt eigentlich nur noch, Isabelle mit den richtigen Parametern zu starten und das Ergebnis wieder einzulesen.

Ich denke, dass die Ausführungsreihenfolge auch für die Implementierungsreihenfolge maßgeblich ist. Aus Testgründen ist es sinnvoll, zuerst die Beweisaussage und danach die Regelanwendungen zu generieren. Um ein Verständnis für mögliche Probleme zu entwickeln, werde ich mir vor der Beschäftigung mit Isabelle das SMTInterpol-Format ansehen.

1.3 Theorien und Vermutungen

Dieser Abschnitt besteht aus meinen Vermutungen bezüglich der Problematik mit den Theorien. Sie basieren auf meinem Wissen aus der Vorlesung *Decision Procedures*. Ob diese Aussagen tatsächlich so zutreffen, soll auch im ersten Schritt bestätigt bzw. widerlegt werden.

1.3.1 Lineare Arithmetik

Die Theorie der linearen Arithmetik benutzt den simplex-basierten *Dutertre-de Moura-Algorithmus*. Eine Formel ϕ hat die Form $\phi \equiv \bigwedge_{j=1}^{m} \sum_{i=1}^{n} a_i x_i \leq b_j$, wobei die a_i, b_j Konstanten und die x_i Variablen sind.

Ein Gegenbeispiel besteht aus den Farkas-Koeffizienten, die einen Widerspruch belegen. Dies sind die Faktoren, mit denen man jede Zeile multiplizieren muss, um am Ende durch Aufsummieren aller Zeilen alle Koeffizienten gleich Null zu erhalten und so auf einen Widerspruch (z.B. $0 \leq -1$) zu kommen.
Implementierung: relativ naheliegend: Multiplikation der Zeilen mit den Koeffizienten, dann Addition der Zeilen. Wahrscheinlich wird eine Regel folgender Form benötigt:

$$\frac{a \leq b}{c \cdot a \leq c \cdot b}$$

Ein Erfüllbarkeitsbeweis gibt eine erfüllende Belegung der Variablen an.
Implementierung: Berechnung und Überprüfung der Ergebnisse.

Ich weiß nicht, wie gut Isabelle zum Rechnen ausgelegt ist. Möglicherweise benötigt man für diese Theorie Isabelle auch gar nicht, da alles nur auf einfaches Rechnen hinausläuft → Klärungsbedarf.

EDIT 1: [Der Grund, warum diese Rechnungen nicht in einem normalen Rechenprogramm durchgeführt werden, ist, dass man dann auf die Korrektheit eines weiteren Programms vertrauen muss.]

1.3.2 Congruence Closure

Die Theorie der Gleichheit wird mit der Congruence Closure entschieden. Eine Formel ϕ hat die Form $\phi \equiv \bigwedge_{i=1}^{m} a_i = b_i \wedge \bigwedge_{j=m+1}^{n} a_j \neq b_j$, wobei die a_i, b_i Terme sind.

Ein Gegenbeispiel benötigt nur eine Ungleichung und die Gleichheitsformeln, die durch Transitivität einen Widerspruch ergeben.
Implementierung: sehr einfach: Anwenden der Kongruenz- und Transitivitätsregel.

Ein Erfüllbarkeitsbeweis gibt ein Modell an, das die Formel erfüllt. Dieses besteht aus Äquivalenzklassen von Termen, die gleich sind. Untereinander sind die Klassen jedoch verschieden.

Implementierung: Isabelle beherrscht womöglich nicht direkt einen Erfüllbarkeitsbeweis. Das kann man umgehen, indem man die Variablen existenzquantifiziert. Um diese Quantoren aufzulösen, benötige ich eventuell ein Universum (z.B. natürliche Zahlen). Möglicherweise benötige ich für die uninterpretierten Funktionen eine Extrabehandlung.

2 Erinnerungsliste

Hier werden alle Dinge aufgelistet, die nicht in Vergessenheit geraten sollen.

1. Verschachtelte $\wedge/\vee$ könnten mit der `blast`-Taktik von Isabelle langsam ablaufen. Möglicherweise muss ich hierfür eine eigene Regel schreiben.
2. Es gibt die `:named`-Annotation. Diese soll später auch unterstützt werden.
3. Auch im Beweis gibt es die Möglichkeit, *let* zu benutzen und somit die Ausrücke zu verkürzen.

3 Zeitplan

Aufgabe	mögliche Probleme	Zeit bis	abgeschlossen
Isabelle-Lemma für eine konkrete LA-Formel generieren	—	5.11.	✓
Isabelle-Lemma für beliebige LA-Formeln generieren	manche Operatoren nicht unterstützt?	9.11.	✓ (außer *let*)
Isabelle-Lemma für beliebige CC-Formeln generieren	uninterpretiere Funktionen	12.11.	✓ (außer *let*)
Isabelle-Theorie für CC-Formeln erstellen	Isabelle-Theorie verstehen	19.11.	
einfachen Beweis für eine konkrete CC-Formel generieren	Isabelle-Lemmata benötigt?	27.12.	
CC-Beweise abschließen	—	1.12.	
Isabelle-Theorie für LA-Formeln erstellen	Isabelle-Theorie verstehen	8.11.	
einfachen Beweis für eine konkrete LA-Formel generieren	Isabelle-Lemmata benötigt?	15.12.	
LA-Beweise abschließen	—	22.12.	

4 Stand

Hier wird der aktuelle Stand festgehalten.

- `main()` liest `test.smt2` ein, erstellt ein neues `ProofChecker`-Objekt und ruft `check()` auf.
  - `ProofChecker`-Konstruktor mit eingelesener `*.smt2`-Datei
- Hauptmethode `check()` arbeitet folgendermaßen:
  - Die Assertions werden in eine große Konjunktion von Formeln umgewandelt, die dann `false` impliziert (da es sich um einen Unerfüllbarkeitsbeweis handelt).
    - Alle SMTLib-Operationen werden übersetzt (außer *let*). Jede Integer- und Real-variable wird außerdem getypt.

- Der Beweis wird noch komplett ignoriert.
- Die fertige Datei wird aktuell unter dem Namen `old.thy` gespeichert, wobei `old` der alte Dateiname ist (inklusive Dateiendung).
- Isabelle wird in Emacs gestartet und öffnet die gespeicherte Datei.
- Wenn Emacs geschlossen wird, gibt die Methode `false` zurück, da sie die Isabelle-Ausgabe noch nicht versteht.