HTWK
Hochschule für Technik,
Wirtschaft und Kultur Leipzig

IM Fakultät
Informatik und Medien

Abschlussarbeit
zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

im Studiengang Informatik
der Fakultät Informatik und Medien

Erfüllbarkeit und Optimierung von Pressenplanungsproblemen mit Constraint-Systemen

Julian Bruder
Leipzig, den 26. August 2024

Erstgutachter: Prof. Dr. Johannes Waldmann
Zweitgutachter: Dr. Sebastian Möbius

Die Pressenplanung von Holzbalken aus Bauteilen erfordert eine fein abgestimmte Konfiguration, um die Pressen optimal auszulasten und den Verschnitt der Balken zu minimieren. Dafür hat die Germanedge Solutions GmbH[1], Industriepartner dieser Arbeit, bereits eine unternehmensinterne Heuristik, welche aufgrund ihrer Natur allerdings möglicherweise (optimale) Lösungen nicht berücksichtigt. Im Rahmen dieser Arbeit wird daher eine prototypische Lösung mithilfe der Methoden der Constraint-Programmierung, speziell jener der Satisfiability Modulo Theories (SMT), erarbeitet, welche optimale Lösungen finden soll. Dazu werden die Constraints der Domäne zunächst mithilfe der Prädikatenlogik (FOL) modelliert und anschließend in einer konkreten Repräsentation des Modells implementiert.

Dabei wird ein Programm entwickelt, welches aus den textuellen Eingaben der Pressen- und Bauteilspezifikationen ein Constraint-System erzeugt, das anschließend von einem externen Constraint-Solver gelöst wird. Das Programm wandelt die Lösung des Solvers folglich in eine, zu den bereits existenten Schnittstellen passende, textuelle Ausgabe um. Weiter ist das Programm so strukturiert, dass verschiedene Logiken aus SMT zur Erzeugung des Constraint-Systems genutzt werden können. Dies wird durch Wahl der Programmiersprache Haskell und deren ausdrucksstarkes Typsystem ermöglicht. Als Grundlage zur Kodierung des Pressenplanungsproblems und Interaktion mit externen Solvern wird die Haskell-Bibliothek `Hasmtlib` [Bru24a] verwendet und an geeigneten Stellen erweitert.

Das Ziel für die prototypische Lösung des Problems der Pressenplanung ist die korrekte, optimale und bestenfalls schnelle (unter sechs Minuten) Lösung bekannter Testfälle. Da es sich hier um eine prototypische Lösung handelt, werden einige Sonderfälle des Pressenplanungsproblems in dieser Arbeit nicht betrachtet. Am Ende wird die Performance der prototypischen Lösung mit jener der aktuell existenten heuristischen verglichen.

[1] https://www.germanedge.com/, abgerufen am 21.08.2024

Inhaltsverzeichnis

1. Einleitung

Die Germanedge Solutions GmbH ist Deutschlands Vorreiter in der Digitalisierung der Produktion und unterstützt Unternehmen mit individuellen Softwarelösungen für die Industrie 4.0 [VAB18]. Darunter auch ein nicht genannter Kunde, welcher Germanedge mit der Optimierung der Planung von Holzpressen beauftragt hat. Dabei werden Holzbauteile zu Holzbalken verleimt und anschließend in Pressen gepresst. Die Art und Weise der Anordnung der Bauteile in den Pressen wird als Pressenplan bezeichnet. Abbildung 1.1 visualisiert einen solchen Pressenplan. Die Presse (grau) presst in Richtung der Pfeile nach unten. Die Balken (gelb) liegen horizontal und bestehen aus Bauteilen, welche durch die vertikalen (schwarzen) Linien getrennt sind. Der Rest jedes Balkens ist rot hervorgehoben.

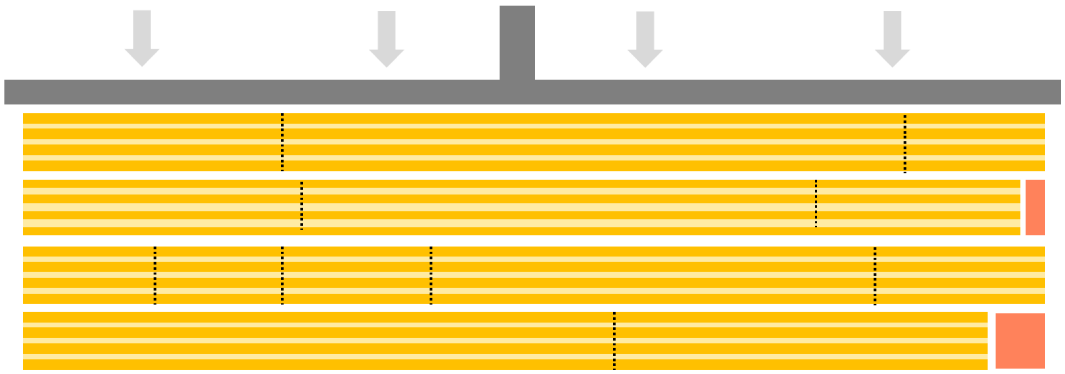

Abbildung 1.1.: Darstellung eines Pressenplans

An einen Pressenplan werden verschiedene Anforderungen wie beispielsweise die Gleichheit der Längen aller Balken in einer Presse gestellt. Neben diesen Anforderungen sind auch einige Optimierungsziele für den Pressenplan von Bedeutung. Darunter zum Beispiel die Minimierung des Rests, der Balken auf die gleiche Länge auffüllt und somit zusätzliche Materialkosten bedeutet.

Germanedge konnte für dieses Problem bereits eine Lösung finden, diese ist allerdings unter Firmenverschluss. Es ist lediglich bekannt, dass jene einen heuristischen Ansatz wählt, was bedeutet, dass (optimale) Lösungen verloren gehen.

Daher wird in dieser Arbeit ein neuer, vollständiger Ansatz zur Lösung des Problems gewählt. Dieser besteht in der Kodierung der Anforderungen an den Pressenplan als Constraints in einem Constraint-System. Ein Constraint ist eine aussagenlogische Formel, welche unter einer erfüllenden Belegung den Wahrheitswert `Wahr` haben muss. Für die dafür in dieser Arbeit verwendeten Satisfiability Modulo Theories (SMT) und deren Sprachstandard SMTLib Version 2.6 [BFT17] beschreibt folgendes Constraint beispielsweise die Neutralität der Null im Monoid $(\mathbb{N}_0, +, 0)$:

```
(assert (forall ((x Int)) (and (= (+ x 0) x) (= (+ 0 x) x))))
```

Listing 1: SMTLib-Kodierung der Neutralität der Null im Monoid $(\mathbb{N}_0, +, 0)$

Ein Constraint-System besteht aus der Konjunktion mehrerer Constraints. In typischen Anwendungsbereichen von SMT wie der Programmverifikation [BT18] werden im Gegensatz zu Beispiel 1 Unbekannte (Variablen) modelliert, für welche der Solver eine erfüllende Belegung unter Beachtung der Constraints ermitteln soll. So könnte obiges Beispiel als Frage nach dem neutralen Element im Monoid $(\mathbb{N}_0, +, x)$ folgend formuliert werden:

```
(declare-fun z () Int)
(assert (forall ((x Int)) (and (= (+ x z) x) (= (+ z x) x))))
```

Listing 2: SMTLib-Kodierung des unbekannten neutralen Elements x im Monoid $(\mathbb{N}_0, +, x)$

Um zum Pressenplanungsproblem zurückzukehren, stellt sich die Frage, wie dieses in SMTLib Version 2.6 kodiert werden kann. Eine Beispieleingabe von Bauteilen könnte folgende sein:

Länge	**Höhe**	**Anzahl**
5345.0	235.0	4
5219.0	287.0	40
5319.0	287.0	36
5220.0	235.0	59
4796.0	287.0	13
5320.0	194.0	48
4740.0	287.0	34

Tabelle 1.1.: Beispieleingabe von Bauteilen, jede Zeile entpricht einer Bauteilspezifikation

Da das Formulieren der Constraints für einen Pressenplan mit gegebenen Bauteilen händisch in der Constraint-Sprache SMTLib sehr schwer- und fehleranfällig ist, erzeugt ein Programm in einer Gastsprache das Constraint-System für die Constraint-Sprache. Für diese Arbeit wird als jene Gastsprache Haskell [Hud+07] gewählt. Mit der Haskell-Bibliothek `Hasmtlib` [Bru24a] lässt sich beispielhaft das bereits erwähnte Constraint gleicher Länge aller Balken in einer Presse folgend formulieren:

```
barLength :: [Bar (Expr RealSort)] -> Expr BoolSort
barLength = all
              (\(b1,b2) -> (b1^.press === b2^.press) ==>
                (b1^.length === b2^.length)
              ) . binom
```

Listing 3: Haskell-Code für das Constraint gleicher Länge aller Balken einer Presse

Die Bibliothek erzeugt dann das Constraint-System in der Constraint-Sprache SMTLib. Folgend ein beliebiger Ausschnitt der SMTLib-Kodierung:

```
(declare-fun var_1 () Real)
(declare-fun var_2 () Real)
...
(assert (= var_34 (not var_33)))
(assert (= var_38 (and var_36 var_37)))
(assert (= var_39 (or var_35 var_38)))
...
```

Listing 4: Ausschnitt der Kodierung eines Pressenplanungsproblems

Die Kodierung wird an einen SMT-Solver gegeben, welcher die Erfüllbarkeit des Problems bestimmt und für den Fall *erfüllbar* eine erfüllende Belegung für die Variablen des Problems ermittelt. Zum Beispiel:

```
1 sat
2
3 (
4   (define-fun var_1 () Real    (- 12.0))
5   (define-fun var_2 () Real    33.4)
6   ...
7 )
```

Listing 5: Ausschnitt des Solver-Outputs der Lösung eines Pressenplanungsproblems

Jene Belegung transformiert das Programm dann in das gewünschte Ausgabeformat:

Presse	**Schicht**	**Position**	**Länge**	**Höhe**
1	1	1	5345.0	235.0
1	1	2	5345.0	235.0
1	1	3	5345.0	235.0
1	2	1	5219.0	287.0
...	...	...	...	...

Tabelle 1.2.: Beispielausgabe des Pressenplans

Die Struktur dieser Arbeit ist folgende:

In Kapitel 2 werden die, für das weitere Verständnis der Arbeit erforderlichen, Grundlagen der Prädikatenlogik erklärt.

Danach wird in Kapitel 3 das Problem der Pressenplanung dargelegt, modelliert und nach, für das Problem relevanten, Zielfunktionen optimiert.

Folgend wird in Kapitel 4 gezeigt, wie Erfüllbarkeitsprobleme der Satisfiability Modulo Theories im Sprachstandard SMTLib Version 2.6 kodiert werden können.

In Kapitel 5 wird zunächst auf wenige, notwendige Grundlagen von Haskell eingegangen, bevor die Kodierung der in Abschnitt 3.2 erstellten Modelle des Pressenplanungsproblems in Haskell erläutert wird. Zusätzlich werden dort einige Erweiterungen der Haskell-Bibliothek `Hasmtlib` diskutiert.

Weiter werden die Ergebnisse der Kodierungen in Kapitel 6 mithilfe von Laufzeitmessungen ausgewertet.

Abschließend wird das Ergebnis dieser Arbeit in Kapitel 7 zusammengefasst.

2. Grundlagen der Prädikatenlogik

Erfüllbarkeit, Terme, Atome, Formeln ... [KB08]

3. Problemdomäne Pressenplanung

In diesem Kapitel wird das Problem der Pressenplanung zunächst erläutert, bevor die prädikatenlogische Modellierung des Problems in zwei Modellen erfolgt. Abschließend werden die Modelle um Optimierungsziele erweitert.

3.1. Anforderungen

Im Pressenplanungsproblem sind drei Entitäten und deren Eigenschaften von elementarer Bedeutung: Pressen, Balken und Bauteile. Diese haben jeweils die Eigenschaften Höhe, Breite und Länge. Zusätzlich gibt es Holzlamellen, welche durch Zersägen eines sogenannten *Endlosbretts* entstehen. Dieses kann von einer Maschine *scheinbar* endlos ausgegeben werden. Höhe und Breite der Lamelle sind fest. Lamellen werden vertikal zu Bauteilen verleimt, sodass Bauteile immer die Form eines Quaders haben. Diese Verleimung ist der eigentliche Grund für die Notwendigkeit des Pressens. Folgend eine Abbildung verleimter Lamellen:

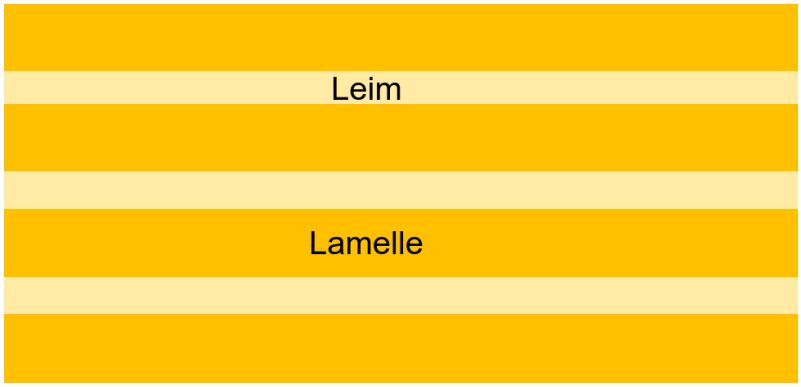

Abbildung 3.1.: Vertikale Verleimung von Lamellen

Bauteile wiederum formen Balken - ebenfalls Quader. Bauteile erhält man durch vertikales Zersägen eines Balkens. Folgend eine Abbildung eines Balkens aus drei Bauteilen mit Rest:

Abbildung 3.2.: Balken aus drei Bauteilen mit Rest, Sägelinien schwarz gepunktet

Balken werden vertikal in Pressen gestapelt und anschließend gepresst. Eine mit Balken befüllte Presse, bei der festgelegt ist wie die Balken zu Bauteilen nach der Pressung zersägt werden müssen, nennt man auch Pressenplan. In einem Pressenplan ist jedem Bauteil höchstens eine Position in einer Schicht (von unten gezählter Balken) einer Presse zugeordnet. Folgend die Visualisierung eines Pressenplans für eine Presse:

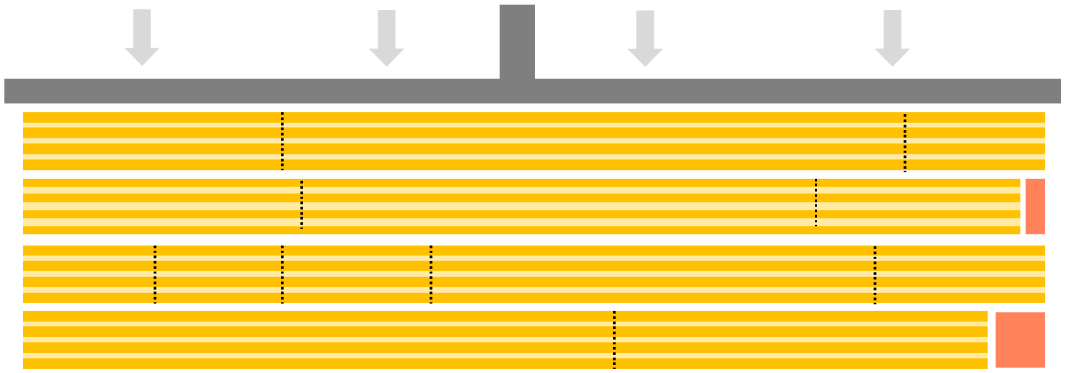

Abbildung 3.3.: Pressenplan für eine Presse mit vier Balken, 13 Bauteilen und zwei Reststücken

Da Pressen im Allgemeinfall in einer Pressung nur Bauteile gleicher Breite pressen können, werden die Bauteile nach Breite vorsortiert. Damit reduziert man die Dimension des Problems vom 3-Dimensionalen auf das 2-Dimensionale, da nur noch Länge und Höhe aller Entitäten berücksichtigt werden muss.

Die Eingabe für das Pressenplanungsproblem ist eine Liste von Bauteilen mit Länge und Höhe sowie deren Anzahl. Diese Eingabe wird auch Auftrag genannt. Die Lösung des Pressenplanungsproblems (Ausgabe) ist ein Pressenplan. An einen Pressenplan bestehen folgende Anforderungen:

- Pressen haben eine Minimal- und Maximalhöhe. Die Summe der Höhe aller Balken einer Presse muss in diesem Intervall liegen.
- Pressen haben eine Minimal- und Maximallänge. Die Länge aller Balken einer Presse muss in diesem Intervall liegen.
- Alle Balken einer Presse haben die gleiche Länge. Dazu kann der Balken so aus Bauteilen zusammengesetzt sein, dass ein Rest übrig bleibt.
- Alle Bauteile, die einen Balken formen, haben die gleiche Höhe.
- Jedes Bauteil eines Auftrages ist höchstens einem Balken zugeordnet.
- Jeder Balken ist genau einer Presse zugeordnet.

3.2. Modellierung

Mathematische Modelle für Erfüllbarkeit des Problems. Warum zwei Modelle? Größe des Problems (Unbekannte, Formeln)?

3.3. Optimierung

Erweiterung der Modelle um Optimierungsziele. OMT, Incremental refinement, ...

4. Satisfiability Modulo Theories

In diesem Kapitel wird zuerst das Erfüllbarkeitsproblem der Satisfiability Module Theories - kurz SMT - vorgestellt. Danach wird auf die Kernelemente des Sprachstandards SMTLib Version 2.6 [BFT17] eingegangen. Dieser ermöglicht die Kodierung von SMT-Problemen für SMT-Solver [BT18]. Ein SMT-Solver - kurz Solver - ist ein Programm, welches für ein kodiertes Problem die Erfüllbarkeit dieses und für den Fall *erfüllbar* (englisch *satisfiable* - kurz *sat*) ein Modell für jenes bestimmen kann. Einige bekannte Solver sind zum Beispiel Z3 [DB08], CVC5 [Bar+22], Yices [Dut14] und OpenSMT [Bru+10]. Diese und viele andere nehmen jährlich an dem internationalen Wettbewerb SMT-COMP [BMS05] für SMT-Solver teil, wo sie sich in verschiedenen Disziplinen miteinander messen. Am Ende dieses Kapitels wird zudem betrachtet, wie SMT-Probleme nach Zielfunktionen optimiert werden können.

4.1. Einführung in das Erfüllbarkeitsproblem SMT

Logiken, Sorten erklären. Welche Logiken hier relevant? Wie lösen die Solver die Probleme der Logiken? Wie performant lösen die Solver diese - Komplexität? Aktuelle SMT-Comp - beste Solver für relevante Kategorien nennen.

4.2. Sprachstandard SMTLib Version 2.6

Standard Version 2.6: [BFT17]

4.3. SMT-Kodierung von Anzahl-Constraints

Oftmals werden zur Kodierung von SMT-Problemen auch Anzahl-Constraints benötigt. Auf einer Menge von n gegeben booleschen Variablen $\{b_0, b_1, \ldots, b_{n-1}\}$ soll dabei die Anzahl wahr belegter Variablen nach k restriktiert werden. Dabei existieren die folgenden drei Restriktionen:

- Mindestens k: $atLeast(k, \{b_0, b_1, \ldots, b_{n-1}\})$
- Genau k: $exactly(k, \{b_0, b_1, \ldots, b_{n-1}\})$
- Höchstens k: $atMost(k, \{b_0, b_1, \ldots, b_{n-1}\})$

Anzahl-Constraints sind nicht Teil des SMTLib-Standards Version 2.6 [BFT17], sodass nur wenige Solver wie Z3 solche Kodierungen direkt unterstützen [Bjø+19]. Daher wird zur manuellen Kodierung of die ITE-Kodierung verwendet [KGK19]. Dabei werden die einzelnen booleschen Unbekannten mithilfe eines If-Then-Else-Ausdrucks (kurz ITE) in numerische Ausdrücke umgewandelt, anschließend summiert und verglichen. Mithilfe der Hilfsfunktion $count(\{b_0, b_1, \ldots, b_{n-1}\})$ lassen sich die Anzahl-Constraints einfach kodieren:

$$count(B) = \sum_{b \in B} \left(\begin{cases} 1 & , b \\ 0 & , \text{sonst} \end{cases} \right)$$

$$atLeast(k, B) = k \leq count(B)$$

$$exactly(k, B) = k = count(B)$$

$$atMost(k, B) = k \geq count(B)$$

Je nach gewählter Logik erfolgt die Summierung in der entsprechenden Sorte. Ist für das SMT-Problem beispielsweise die Logik QF_LRA gewählt, dann summiert $count(B)$ Terme der Sorte `Real`. Dabei entstehen für jedes Anzahl-Constraint offensichtlich n ITE-Ausdrücke, $n - 1$ binäre Additionen und ein Vergleich.

4.4. Optimierung - Optimization Modulo Theories

nuZ3 Paper hier sehr hilfreich [BPF15]

5. Implementierung in Haskell

In diesem Kapitel wird gezeigt, wie SMT-Probleme mit funktionaler Programmierung in SMTLib kodiert werden können. Dazu werden zunächst einige Grundlagen der funktionalen Programmiersprache Haskell erklärt. Danach wird der Kern der Funktionsweise der Haskell-Bibliothek `Hasmtlib` [Bru24a] untersucht. Ferner werden einige geeignete Erweiterungen dieser diskutiert. Anschließend wird mithilfe von Quellcode-Ausschnitten demonstriert, wie das Pressenplanungsproblem polymorph in der SMT-Sorte in Haskell implementiert werden kann. Zum Schluss wird die Validierung vermeintlicher Lösungen betrachtet.

5.1. Funktionale Programmierung mit Haskell

Was ist ein ADT? Polymorphismus. Pattern-Matching. Was ist ein GADT und was kann er besser als ein ADT? Typ- und Kind-System anreißen - Phasentrennung. Typklassen. Typeoperators. Was ist eine pure Funktion? Wie kann man Nebenwirkungen dennoch modellieren - was ist eine Monade? Was ist Lazy? Speicher-Aspekte: Referentielle Integrität, ...

5.2. Hasmtlib - Haskell-Bibliothek für SMTLib

Die Haskell-Bibliothek `Hasmtlib` [Bru24a] ist eine eingebettete domänenspezifische Sprache (englisch *embedded domain-specific language* - kurz *eDSL*) für den SMTLib-Standard Version 2.6 [BFT17] in Haskell. Eine eDSL ist eine, in einer Gastsprache eingebettete, anwendungsspezifische Sprache, welche auf Grundkonstrukten der Gastsprache aufbaut und somit den Implementierungsaufwand einer eDSL erheblich reduziert [SCK04].

Haskell wird aufgrund seines starken Typsystems oft als Gastsprache für solche eingebetteten Sprachen verwendet [Gil14]. Listing 6 zeigt beispielsweise, wie `Hasmtlib` SMTLib-Ausdrücke als abstrakten Syntaxbaum mit einem GADT repräsentiert.

```haskell
data SMTSort =  BoolSort | IntSort | RealSort | ...

data Expr (t :: SMTSort) where
  Var       :: SMTVar t -> Expr t
  Constant  :: Value  t -> Expr t
  Plus      :: Num (HaskellType t) => Expr t -> Expr t -> Expr t
  ...
  IDiv      :: Expr IntSort  -> Expr IntSort  -> Expr IntSort
  Div       :: Expr RealSort -> Expr RealSort -> Expr RealSort
  LTH       :: (Ord (HaskellType t), KnownSMTSort t) =>
                                    Expr t -> Expr t -> Expr BoolSort
  ...
  Not       :: Boolean (HaskellType t) => Expr t -> Expr t
  And       :: Boolean (HaskellType t) => Expr t -> Expr t -> Expr t
  ...
  Sqrt      :: Expr RealSort -> Expr RealSort
```

Listing 6: Repräsentation von SMTLib-Ausdrücken mithilfe eines GADTs in `Hasmtlib` [Bru24b]

Dabei ist der GADT `Expr t` polymorph in der `SMTSort`, sodass dessen Kind `Expr t :: SMTSort -> Type` ist. Da Werte des Datentyps `SMTSort` hier als Typen verwendet werden, nennt man `SMTSort` auch *Data-Kind* oder *Promoted-Type* [EW12]. Besonders relevant dafür ist das in 5.1 erklärte *Type-Refinement*. So weist beispielsweise ein Pattern-Match auf den Konstruktor `Sqrt` bereits zur Übersetzungszeit nach, dass `t ~ RealSort` gilt. Bei anderen Konstruktoren wie `And` kann die Belegung des `t` in `Expr t` mithilfe von Singletons ermittelt werden. Ein *Singleton* ist ein Laufzeit-Zeuge (Wert) für genau einen Übersetzungszeit-Typen [EW12]. Diese Abhängigkeit ist aufgrund Haskell's strenger Phasentrennung notwendig und ermöglicht eine *dependent*-ähnliche Kodierung wie in den Dependently-Typed Programmiersprachen Idris [BRA13] und Agda [Nor09].

Jeder Typ vom Kind `SMTSort` erhält also einen Singleton (Wert). Wie in Listing 7 zu sehen, stellt `Hasmtlib` den Singleton-Typen `SSMTSort` ebenfalls als GADT dar.

```haskell
data SSMTSort (t :: SMTSort) where
  SIntSort    :: SSMTSort IntSort
  SRealSort   :: SSMTSort RealSort
  SBoolSort   :: SSMTSort BoolSort
  ...
```

Listing 7: Repräsentation des Singleton-Typs SSMTSort in `Hasmtlib` [Bru24c]

Die Verbindung zwischen Typ und Wert wird mit der in Listing 8 dargestellten Typklasse `KnownSMTSort` erreicht.

```haskell
class    KnownSMTSort (t :: SMTSort) where sortSing :: SSMTSort t
instance KnownSMTSort IntSort        where sortSing = SIntSort
instance KnownSMTSort RealSort       where sortSing = SRealSort
instance KnownSMTSort BoolSort       where sortSing = SBoolSort
...
```

Listing 8: Typklasse zur Verbindung von Typ SMTSort zu Wert SSMTSort in `Hasmtlib` [Bru24c]

Mit den genannten Mechanismen garantiert `Hasmtlib` die ausschließliche Erzeugung von sauber typisierten (englisch *well-typed*) Ausdrücken [Pey+06]. Diese werden anschließend von der Bibliothek in SMTLib-Syntax übersetzt und an einen SMT-Solver gegeben. Dessen Antwort parst und übersetzt `Hasmtlib` dann zurück in die entsprechenden Haskell-Typen. Folgend ein kleines Programm zur Erläuterung:

```haskell
module Example.Foo where

import Language.Hasmtlib
import GHC.Generics

-- a custom data-type marshalled to a solver
data Foo a = Foo a a deriving (Show, Generic)
instance Codec a => Codec (Foo a)
instance Variable a => Variable (Foo a)

main :: IO ()
main = do
  result <- solveWith @SMT (solver cvc5) $ do
    -- set logic for solver to use
    setLogic "QF_LIA"

    -- construct a Foo with variables
    foo@(Foo l r) :: Foo (Expr IntSort) <- variable

    -- constrain foo
    assert $ l === r + r
    assert $ r === 42

    return foo

  print result

-- Prints: (Sat,Just (Foo 84 42))
```

Listing 9: Beispielhafte Verwendung von Hasmtlib

Das Beispiel in Listing 9 demonstriert außerdem die pure Formelkonstruktion in `Hasmtlib`. Der Term `r + r` in Zeile 21 hat den Typ `Expr IntSort` während das

Atom `l === r + r` den Typ `Expr BoolSort` hat. Lediglich Variablenerzeugung und Formel-Assertion sind monadisch. Dadurch lassen sich viele Funktionalitäten der Gastsprache wiederverwenden, zum Beispiel die Funktion `sum` aus `Data.Foldable`: `sum :: (Foldable t, Num a) => t a -> a`.

5.2.1. Ergänzung um Anzahl-Constraints

`Hasmtlib` unterstützte bis zum Zeitpunkt dieser Arbeit keine Anzahl-Constraints. Daher werden diese im Rahmen dieser Arbeit der Bibliothek hinzugefügt. Die Implementierung erfolgt analog zur in 4.3 präsentierten ITE-Kodierung:

```
count' :: forall t f. (Functor f, Foldable f, Num (Expr t))
    => Proxy t -> f (Expr BoolSort) -> Expr t
count' _ = sum . fmap (\b -> ite b 1 0)

count :: forall t f. (Functor f, Foldable f, Num (Expr t))
    => f (Expr BoolSort) -> Expr t
count = count' (Proxy @t)

atMost  :: forall t f. (Functor f, Foldable f, Num (Expr t), Orderable (Expr t))
    => Expr t -> f (Expr BoolSort) -> Expr BoolSort
atMost  k = (<=? k) . count

atLeast :: forall t f. (Functor f, Foldable f, Num (Expr t), Orderable (Expr t))
    => Expr t -> f (Expr BoolSort) -> Expr BoolSort
atLeast k = (>=? k) . count

exactly :: forall t f. (Functor f, Foldable f, Num (Expr t), Orderable (Expr t))
    => Expr t -> f (Expr BoolSort) -> Expr BoolSort
exactly k = (=== k) . count
```

Listing 10: ITE-Kodierung von Anzahl-Constraints in `Hasmtlib` [Bru24d]

5.2.2. Ergänzung um Incremental Refinement

5.2.3. Ergänzung um Observable-Sharing

Observable Sharing motivieren. Grundlagen für Implementierung erklären: Paramorphismen (sprengt das den Rahmen?), StableNames, Tseitin. Ausschnitte der Implementierung zeigen? Warum der Aufwand? Bringt das überhaupt was für den Anwendungsfall Pressenplanung? Wenn nein: Gehört das hier dann überhaupt rein? Also benchmarken...? Geht es vielleicht noch besser als (DAG-)Speicher-Sharing? Vergleich Formeln auf Gleichheit + Umformungen ermöglichen viel mehr Sharing (speicher-unabhängig). Warum wird das noch nicht gemacht?

5.3. Kodierung des Pressenplanungsproblems

Zwei Modelle erschweren hier möglicherweise Verständnis für den Leser. Für ein Modell entscheiden: Relationales Modell begründend wegwerfen, auf Laufzeitmessung verweisen. Ebenso Unterscheidung Incremental Refinement/OMT, OMT wegwerfen, weil Laufzeit. Geeignete Programmauszüge: Polymorphe Kodierung in Sorte (damit Logik), vereinzelt Datentypen und Constraints analog zu Modellierung. Minimal IO erklären.

5.4. Validierung der Lösung

Warum validieren? Wie validieren? Wenn Modell richtig und auch richtig in SMT kodiert, dann Ergebnis richtig gdw. Solver Spezifikation erfüllt. Erfüllt Solver die Spezifikation? Ja hoffentlich - das müssen wir denen abkaufen - oder Nachweis bspw. per SMT-Comp?

6. Auswertung und Laufzeitmessung

Was haben wir für Testfälle? Woher kommen die Testfälle? Wie wurden die Testfälle ausgewählt? Sind sie repräsentativ für reale Probleme? Vorgehen Laufzeitmessung erwähnen. Maschinen-Specs nennen.

6.1. Vergleich mit realistischem Timeout

Time-Limit 5 Minuten, Vergleich Heuristik Germanedge vs SMT. Nur ein Modell oder beide? - Beide. Auch OMT? - Ja. Auswertung der Ergebnisse.

6.2. Vergleich mit liberalem Timeout

Wer findet bessere Lösung, wenn Timeout sehr liberal (24h?)? Hier werden Hardware-Specs ein Problem. Das muss auf Pool-PCs der HTWK gemacht werden. Darf da ein Binary von der Heuristik laufen? Auswertung der Ergebnisse.

7. Zusammenfassung

Literatur

[Bar+22] Haniel Barbosa u. a. „cvc5: A Versatile and Industrial-Strength SMT Solver“. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Hrsg. von Dana Fisman und Grigore Rosu. Cham: Springer International Publishing, 2022, S. 415–442. ISBN: 978-3-030-99524-9.

[BFT17] Clark Barrett, Parcal Fontaine und Cesare Tinelli. *The SMT-LIB Standard Version 2.6*. Techn. Ber. Abgerufen am: 21.08.2024. Department of Computer Science, The University of Iowa, 2017. URL: https://smt-lib.org/papers/smt-lib-reference-v2.6-r2017-07-18.pdf.

[Bjø+19] Nikolaj Bjørner u. a. „Programming Z3“. In: Springer Nature, Apr. 2019, S. 148–201.

[BMS05] Clark Barrett, Leonardo de Moura und Aaron Stump. „SMT-COMP: Satisfiability Modulo Theories Competition“. In: *Computer Aided Verification*. Hrsg. von Kousha Etessami und Sriram K. Rajamani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, S. 20–23. ISBN: 978-3-540-31686-2.

[BPF15] Nikolaj S. Bjørner, Anh-Dung Phan und Lars Fleckenstein. „νZ - An Optimizing SMT Solver“. In: *International Conference on Tools and Algorithms for Construction and Analysis of Systems*. 2015. URL: https://api.semanticscholar.org/CorpusID:17542710.

[BRA13] EDWIN BRADY. „Idris, a general-purpose dependently typed programming language: Design and implementation“. In: *Journal of Functional Programming* 23 (Sep. 2013). DOI: 10.1017/S095679681300018X.

[Bru+10] Roberto Bruttomesso u. a. „The OpenSMT Solver“. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Hrsg. von Javier Esparza und Rupak Majumdar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, S. 150–153. ISBN: 978-3-642-12002-2.

[Bru24a] Julian Bruder. *hasmtlib: A monad for interfacing with external SMT solvers*. Version 2.3.2. Abgerufen am: 20.08.2024. 2024. URL: https://hackage.haskell.org/package/hasmtlib.

[Bru24b] Julian Bruder. *hasmtlib: A monad for interfacing with external SMT solvers*. Version 2.4.0. Abgerufen am: 25.08.2024. 2024. URL: https://hackage.haskell.org/package/hasmtlib-2.4.0/docs/src/Language.Hasmtlib.Type.Expr.html#Expr.

[Bru24c] Julian Bruder. *hasmtlib: A monad for interfacing with external SMT solvers*. Version 2.4.0. Abgerufen am: 25.08.2024. 2024. URL: https://hackage.haskell.org/package/hasmtlib-2.4.0/docs/src/Language.Hasmtlib.Type.SMTSort.html#SSMTSort.

[Bru24d] Julian Bruder. *hasmtlib: A monad for interfacing with external SMT solvers*. Version 2.5.0. Abgerufen am: 26.08.2024. 2024. URL: https://hackage.haskell.org/package/hasmtlib-2.5.0/docs/src/Language.Hasmtlib.Counting.html#count%27.

[BT18] Clark Barrett und Cesare Tinelli. „Satisfiability Modulo Theories“. In: *Handbook of Model Checking*. Hrsg. von Edmund M. Clarke u. a. Cham: Springer International Publishing, 2018, S. 305–343. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_11. URL: https://doi.org/10.1007/978-3-319-10575-8_11.

[DB08] Leonardo De Moura und Nikolaj Bjørner. „Z3: an efficient SMT solver“. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, S. 337–340. ISBN: 3540787992.

[Dut14] Bruno Dutertre. „Yices 2.2“. In: *Computer Aided Verification*. Hrsg. von Armin Biere und Roderick Bloem. Cham: Springer International Publishing, 2014, S. 737–744. ISBN: 978-3-319-08867-9.

[EW12] Richard A. Eisenberg und Stephanie Weirich. „Dependently typed programming with singletons“. In: *Proceedings of the 2012 Haskell Symposium*. Haskell '12. Copenhagen, Denmark: Association for Computing Machinery, 2012, S. 117–130. ISBN: 9781450315746. DOI: 10.1145/2364506.2364522. URL: https://doi.org/10.1145/2364506.2364522.

[Gil14] Andy Gill. „Domain-specific Languages and Code Synthesis Using Haskell: Looking at embedded DSLs“. In: *Queue* 12.4 (Apr. 2014), S. 30–43. ISSN: 1542-7730. DOI: 10.1145/2611429.2617811. URL: https://doi.org/10.1145/2611429.2617811.

[Hud+07] Paul Hudak u. a. „A history of Haskell: Being lazy with class“. In: Juni 2007, S. 1–55. DOI: 10.1145/1238844.1238856.

[KB08] Uwe Kastens und Hans Kleine Büning. *Modellierung*. 2., überarbeitete und erweiterte Auflage. München: Carl Hanser Verlag GmbH & Co. KG, 2008. DOI: 10.3139/9783446417212. eprint: https://www.hanser-elibrary.com/doi/pdf/10.3139/9783446417212. URL: https://www.hanser-elibrary.com/doi/abs/10.3139/9783446417212.

[KGK19] Gergely Kovásznai, Krisztián Gajdár und Laura Kovács. „Portfolio SAT and SMT Solving of Cardinality Constraints in Sensor Network Optimization“. In: *2019 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. 2019, S. 85–91. DOI: 10.1109/SYNASC49474.2019.00021.

[Nor09] Ulf Norell. „Dependently Typed Programming in Agda“. In: Jan. 2009, S. 1–2. ISBN: 978-3-642-04651-3. DOI: 10.1007/978-3-642-04652-0_5.

[Pey+06] Simon Peyton Jones u. a. „Simple Unification-based Type Inference for GADTs“. In: *International Conference on Functional Programming (ICFP)*. ACM, 2006. DOI: 10.1145/1159803.1159811.

[SCK04] Sean Seefried, Manuel Chakravarty und Gabriele Keller. „Optimising Embedded DSLs Using Template Haskell“. In: *Generative Programming and Component Engineering*. Hrsg. von Gabor Karsai und Eelco Visser. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, S. 186–205. ISBN: 978-3-540-30175-2.

[VAB18] Saurabh Vaidya, Prashant Ambad und Santosh Bhosle. „Industry 4.0 – A Glimpse“. In: *Procedia Manufacturing* 20 (2018). 2nd International Conference on Materials, Manufacturing and Design Engineering (iCMMD2017), 11-12 December 2017, MIT Aurangabad, Maharashtra, INDIA, S. 233–238. ISSN: 2351-9789. DOI: https://doi.org/10.1016/j.promfg.2018.02.034. URL: https://www.sciencedirect.com/science/article/pii/S2351978918300672.

Appendix

A. Beispiel 1

⋮

B. Beispiel n

Danksagung

Zuallererst möchte ich mich bei Herrn Prof. Dr. Waldmann für das im Wintersemester 2023/2024, in der Vorlesung Constraint-Programmierung geweckte Interesse an jener bedanken. Vielen Dank auch für Ihre zahlreichen Hinweise und Vorschläge während dieser Bachelorarbeit. Weiter möchte ich meinen Dank der Germanedge Solutions GmbH und dort besonders Christoph Schumacher aussprechen, die diese Bachelorarbeit ermöglicht haben. Abschließend auch ein großes Dankeschön an Dr. Sebastian Möbius, welcher mir als Firmenbetreuer jederzeit alle Fragen bestens beantwortet hat.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Die den verwendeten Quellen und Hilfsmitteln wörtlich oder sinngemäß entnommenen Stellen sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Mir sind die strafrechtlichen Konsequenzen einer falschen eidesstattlichen Erklärung bekannt.

Leipzig, den 26. August 2024