Quantifier Heuristics Library

Thomas Tuerk

March 28, 2014

Contents

1 Introduction

1.1 Motivation

In interactive proofs it is often useful to be able to find instantiations for quantifiers. The unwind library[1] allows instantiations of "trivial" quantifiers:

$$\forall x_1 \ \ldots x_i \ldots x_n.\ P_1 \wedge \ldots \wedge x_i = c \wedge \ldots \wedge P_n \Longrightarrow Q$$

and

$$\exists x_1 \ \ldots x_i \ldots x_n.\ P_1 \wedge \ldots \wedge x_i = c \wedge \ldots \wedge P_n$$

can be simplified by instantiating x_i with c. Because unwind-conversions are part of `bool_ss`, they are used with nearly every call of the simplifier and often simplify proofs considerably. However, the unwind library can only handle these common cases. If the term structure is only slightly more complicated, it fails. For example, $\exists x.\ P(a) \Longrightarrow (x = 2) \wedge Q(x)$ cannot be tackled.

There is also the satisfy library[2], which uses unification to show existentially quantified formulas. It can handle problems like $\exists x.\ P_1(x, c_1)\ \wedge \ldots P_n(x, c_n)$ if given theorems of the form $\forall x\ c.\ P_i(x, c)$. This is often handy, but still rather limited.

The quantifier heuristics library[3] provides more power and flexibility. It can handle all the examples that the unwind and satisfy libraries can handle. A few simple examples of what it can do are shown in Table 1.1. Besides the power demonstrated by these examples, the library is highly flexible as well. At it's core, there is a modular, syntax driven search for instantiation. This search consists of a collection of interleaved heuristics. Users can easily configure existing heuristics and add own ones. Thereby, it is easy to teach the library about new predicates, logical connectives or datatypes.

1.2 Structure

This presentation of the quantifier heuristics library is structured into two parts. The first part presents the logical foundations, whereas the second part presents the HOL 4 interface. Readers not interested in the foundations, might skip the first part.

The first part starts with an introduction of the concept of guesses in Section 2.1. Then, Section 2.2 shows how guesses can be derived for interesting terms like equations. These guesses form the base cases of the search for instantiations. Then, Section 2.3

[1] see `src/simp/src/Unwind.sml`
[2] see `src/simp/src/Satisfy.sml`
[3] see `src/quantHeur/quantHeuristicsLib.sml`

Problem	**Result**
basic examples	
$\exists x.\ x = 2 \wedge P(x)$	$P(2)$
$\forall x.\ x = 2 \Longrightarrow P(x)$	$P(2)$
solutions and counterexamples	
$\exists x.\ x = 2$	*true*
$\forall x.\ x = 2$	*false*
complicated nestings of standard operators	
$\exists x_1 . \forall x_2.\ (x_1 = 2) \wedge P(x_1, x_2)$	$\forall x_2.\ P(2, x_2)$
$\exists x_1, x_2.\ P_1(x_2) \Longrightarrow (x_1 = 2) \wedge P(x_1, x_2)$	$\exists x_2.\ P_1(x_2) \Longrightarrow P(2, x_2)$
$\exists x.\ ((x = 2) \vee (2 = x)) \wedge P(x)$	$P(2)$
exploiting unification	
$\exists x.\ (f(8+2) = f(x+2)) \wedge P(f(10))$	$P(f(10))$
$\exists x.\ (f(8+2) = f(x+2)) \wedge P(f(x+2))$	$P(f(8+2))$
$\exists x.\ (f(8+2) = f(x+2)) \wedge P(f(x))$	- (no instantiation found)
partial instantiation for datatypes	
$\forall p.\ c = \mathsf{FST}(p) \Longrightarrow P(p)$	$\forall p_2.\ P(c, p_2)$
$\forall x.\ \mathsf{IS\_NONE}(x) \vee P(x)$	$\forall x'.\ P(\mathsf{SOME}(x'))$
$\forall l.\ l \neq [\,] \Longrightarrow P(l)$	$\forall hd, tl.\ P(hd :: tl)$
context	
$P_1(c) \Longrightarrow \exists x.\ P_1(x) \vee P_2(x)$	*true*
$P_1(c) \Longrightarrow \forall x.\ \neg P_1(x) \wedge P_2(x)$	$\neg P_1(c)$
$(\forall x.\ P_1(x) \Rightarrow (x = 2)) \Longrightarrow (\forall x.\ P_1(x) \Rightarrow P_2(x))$	$(\forall x.\ P_1(x) \Rightarrow (x = 2)) \Rightarrow (P_1(2) \Rightarrow P_2(2))$
$\big((\forall x.\ P_1(x) \Rightarrow P_2(x)) \wedge P_1(2)\big) \Longrightarrow \exists x.\ P_2(x)$	*true*

Table 1.1: Examples

discusses how these guesses are lifted over the term structure. Other concepts used for the search are discussed in Section 2.4. In order to illustrate this abstract presentation, Section 2.5 shows how guess-search works for simple examples.

The second part starts with a high level presentation of the user-interface in Section 3.1. An important concept of the interface are quantifier heuristic parameters, which allow configuring the behaviour of the library. In Section 3.2 first predefined parameters are presented. Then it is demonstrated using concrete examples, how own parameters can be developed.

The presentation closes with a short conclusion.

2 Theoretical Foundations

2.1 Guesses

2.1.1 Weak Guesses for Existential Quantifiers

In the introduction, it is stated that the core component of this work is a search based on heuristics. Abstractly, given a term $\exists x.\ P(x)$ we want to find an instantiation $\lambda fv.i(fv)$ such that $\exists x.\ P(x) \Longleftrightarrow \exists fv.\ P(i(fv))$ holds. In the following, such an i is called a *guess* for variable x in $P(x)$. In order to justify such a guess, we define special predicates that capture the intended semantics.

Definition 2.1.1 (Existential Guess)**.** i is an *existential guess* for P (denoted by $G_\exists(i, P)$) if and only if the equivalence $\exists x.\ P(x) \Longleftrightarrow \exists fv.\ P(i(fv))$ holds.

Example 2.1.2. $G_\exists(K\ 2, \lambda x.\ x = 2)$ holds, because $\exists x.\ x = 2 \Longleftrightarrow 2 = 2$ holds. More interestingly, $G_\exists(\lambda(x, xs').\ x :: xs'), \lambda xs.\ xs \neq [] \wedge P(xs))$ holds. Therefore, $\exists xs.\ xs \neq [] \ \wedge\ P(xs)$ can be simplified to $\exists x\ xs'.\ x :: xs' \neq [] \ \wedge\ P(x :: xs')$.

So, when trying to process $\exists x.\ P(x)$, we search for an existential guess i such that $G_\exists(i, \lambda x.P(x))$ holds. If such a guess is found, the original term can be simplified to $\exists fv.\ P(i(fv))$, which is really *simpler* provided sensible heuristics for creating guesses are used. In particular, guesses i that do not depend on fv frequently occur in practice. The quantifier disappears completely in this common case.

2.1.2 Strong Guesses for Existential Quantifiers

The general idea for coming up with a guess $G_\exists(i, P)$ is preforming a bottom-up search of the syntax tree of P. Unluckily, $G_\exists$ is not well suited for such a search, because it is too weak to easily lift guesses for subterms. Consider, for example, terms of the form $\exists x.\ P_1(x) \vee P_2(x)$. If we have a guess $G_\exists(i, P_1)$, we unluckily can't lift it to a guess $G_\exists(i, \lambda x.\ P_1(x) \vee P_2(x))$ in general. A counterexample is $P_1(x) := \mathit{false}$, $P_2(x) := (x = 2)$ and $i := K\ 3 = \lambda fv.\ 3$. Therefore, stronger types of guesses are introduced that work well with lifting:

Definition 2.1.3 (Point Existenstial Guess)**.** i is a *point existential guess* for P (denoted by $G^\bullet_\exists(i, P)$) if and only if $\forall fv.\ P(i(fv))$ holds.

Example 2.1.4. $G^\bullet_\exists(K\ 2, \lambda x.\ x = 2)$ holds, because $2 = 2$ holds. In addition to weak existential guesses $G_\exists(i, P)$, point existential guesses $G^\bullet_\exists(i, P)$ carry information about

the point i: the point i satisfies P. For example $G^{\bullet}_{\exists}(\lambda(x, xs').\ x :: xs',\ \lambda xs.\ xs \neq [])$ means that $\lambda xs.\ xs \neq []$ holds for all points of the form $x :: xs'$ (for arbitrary x, xs').

Definition 2.1.5 (Gap Existential Guess)**.** i is a *gap existential guess* for P (denoted by $G^{\bar{}}_{\exists}(i, P)$) if and only if $\forall v.\ \Big(\big(\forall fv.\ v \neq i(fv)\big) \Longrightarrow \neg P(v)\Big)$ holds.

Example 2.1.6. In addition to weak existential guesses $G_{\exists}(i, P)$, gap existential guesses $G^{\bar{}}_{\exists}(i, P)$ carry information about all points except the gap i. The guess $G^{\bar{}}_{\exists}(K\ 2, \lambda x.\ x = 2 \wedge Q(x))$ holds, because $\lambda x.\ (x = 2) \wedge Q(x)$ does not hold for all values except possibly 2. Since nothing is known about Q, we don't know, whether P holds for the point 2. This point is a gap in the argument.

Point and gap existential guesses are stronger than existential guesses, i. e. $G^{\bullet}_{\exists}(i, P)$ and $G^{\bar{}}_{\exists}(i, P)$ imply $G_{\exists}(i, P)$ for all i, P. Therefore, they can also be used to instantiate existential quantifiers. Moreover, they allow lifting. For example, the rule $G^{\bullet}_{\exists}(i, P_1) \Longrightarrow G^{\bullet}_{\exists}(i, \lambda x.\ P_1(x) \vee P_2(x))$ holds.

2.1.3 Guesses for Universal Quantifiers

Existential and universal quantification are closely related to each other by negation. In order to lift guesses over negation and also in order to be able to instantiate universal quantifiers, it makes sense to define corresponding guesses for universal quantification exploiting this duality.

Definition 2.1.7 (Universal Guess)**.** i is an *universal guess* for P (denoted by $G_{\forall}(i, P)$) if and only if the equivalence $\forall x.\ P(x) \Longleftrightarrow \forall fv.\ P(i(fv))$ holds.

Definition 2.1.8 (Point Universal Guess)**.** i is a *point universal guess* for P (denoted by $G^{\bullet}_{\forall}(i, P)$) if and only if $\forall fv.\ \neg\ P(i(fv))$ holds.

Definition 2.1.9 (Gap Universal Guess)**.** i is a *gap universal guess* for P (denoted by $G^{\bar{}}_{\forall}(i, P)$) if and only if $\forall v.\ \Big(\big(\forall fv.\ v \neq i(fv)\big) \Longrightarrow P(v)\Big)$ holds.

Similar to guesses for existential quantification, we are mainly interested in weak universal guesses. Point and gap universal guesses are stronger than universal ones and allow lifting.

2.1.4 Overview

Above, 6 types of guesses are formally introduced. Informally, they can be described by the following table:

i **is a ... guess for** P		**intuition**
(weak) existential	$(G_{\exists})$	$\exists x.\ P(x)$ can be safely instantiated with i
(weak) universal	$(G_{\forall})$	$\forall x.\ P(x)$ can be safely instantiated with i
point existential	$(G^{\bullet}_{\exists})$	$P(i)$ holds
point universal	$(G^{\bullet}_{\forall})$	$\neg P(i)$ holds
gap existential	$(G^{\bar{}}_{\exists})$	all v except possibly i do not satisfy P
gap universal	$(G^{\bar{}}_{\forall})$	all v except possibly i satisfy P

This informal description also shows the relative strength of these guesses.

Lemma 2.1.10 (Strength of Guesses)**.** *Point and gap guesses are stronger than weak guesses. This means that the following implications hold for all i and P:*

$$\begin{array}{llll} G_{\dot{\exists}}(i,P) & \Longrightarrow & G_{\exists}(i,P) & \qquad G_{\overset{\bullet}{\exists}}(i,P) \Longrightarrow G_{\exists}(i,P) \\ G_{\dot{\forall}}(i,P) & \Longrightarrow & G_{\forall}(i,P) & \qquad G_{\overset{\bullet}{\forall}}(i,P) \Longrightarrow G_{\forall}(i,P) \end{array}$$

Point and gap guesses cannot be compared to each other in general.

So, the core of the framework searches for guesses. Once found, the guesses are used as described by the following theorem:

Theorem 2.1.11 (Usage of Guesses)**.** *Guesses are used to simplify formulae with quantifiers in various ways. In the most basic case, they can be used to (partially) instantiate quantifiers as follows:*

$$\begin{array}{lll} G_{\exists}(i,P) & \Longrightarrow & \left(\exists x.\ P(x) \ \Leftrightarrow \ \exists fv.\ P(i(fv)) \right) \\ G_{\forall}(i,P) & \Longrightarrow & \left(\forall x.\ P(x) \ \Leftrightarrow \ \forall fv.\ P(i(fv)) \right) \end{array}$$

If the guess does not depend on a free variable (as denoted by $K\ i_c$), the quantifier can be removed completely:

$$\begin{array}{lll} G_{\exists}(K\ i_c,P) & \Longrightarrow & \left(\exists x.\ P(x) \ \Leftrightarrow \ P(i_c) \right) \\ G_{\forall}(K\ i_c,P) & \Longrightarrow & \left(\forall x.\ P(x) \ \Leftrightarrow \ P(i_c) \right) \end{array}$$

Point guesses lead to even more simplification:

$$\begin{array}{lll} G_{\overset{\bullet}{\exists}}(i,P) & \Longrightarrow & \left(\exists x.\ P(x) \ \Leftrightarrow \ \mathit{true} \right) \\ G_{\overset{\bullet}{\forall}}(i,P) & \Longrightarrow & \left(\forall x.\ P(x) \ \Leftrightarrow \ \mathit{false} \right) \end{array}$$

In contrast, the additional strength of gap guesses is mainly important for lifting. For instantiating quantifiers the same rules as for existential and universal guesses apply. However, gap existential guesses are useful for handling unique existential quantification, provided the guess does not contain free variables.

$$\begin{array}{lll} G_{\exists}(K\ i_c,P) & \Longrightarrow & \left(\exists!x.\ P(x) \ \Leftrightarrow \ (P(i_c) \ \wedge \ \forall v.\ P(v) \Rightarrow v = i_c) \right) \\ G_{\dot{\exists}}(K\ i_c,P) & \Longrightarrow & \left(\exists!x.\ P(x) \ \Leftrightarrow \ P(i_c) \right) \\ G_{\overset{\bullet}{\exists}}(K\ i_c,P) & \Longrightarrow & \left(\exists!x.\ P(x) \ \Leftrightarrow \ (\forall v.\ P(v) \Rightarrow v = i_c) \right) \end{array}$$

2.1.5 Oracle Guesses

So far, we discussed guesses that carry semantic information and allow proving equivalences. These are the ones interesting from a theoretical point of view. For implementing an actual tool, guesses without justification are interesting as well. The HOL 4 implementation supports also *oracle guesses*.

Definition 2.1.12 (Oracle Guess)**.** i is an *oracle guess* for P (denoted by $G_?(i, P)$) if the user says so. This means that $G_?(i, P)$ holds for all i and P.

An oracle guess carries no semantic information, it just states something like "*Trust me! This is a sensible guess!*". Therefore, oracle guesses can only be used to prove implications instead of equivalences.

Lemma 2.1.13 (Usage of Oracle Guesses)**.**

$$\begin{array}{lcl} G_?(i,P) & \Longrightarrow & \left(\begin{array}{lcl} \exists x.\ P(x) & \Leftarrow & \exists fv.\ P(i(fv)) \end{array} \right) \\ G_?(i,P) & \Longrightarrow & \left(\begin{array}{lcl} \forall x.\ P(x) & \Rightarrow & \forall fv.\ P(i(fv)) \end{array} \right) \end{array}$$

Oracle guesses allow the user to instantiate quantifiers without formal justification. Therefore, they require hardly any theoretical foundation and are mainly interesting for tool implementation. Therefore, they won't be discussed much below.

2.2 Base Guesses

After introducing guesses, guess-search can be presented now. This presentation is twofold. In this section, guesses for basic terms are discussed. Lifting guesses is presented in the next section.

2.2.1 Equations

The most basic, but also most common source of guesses are equations:

Lemma 2.2.1 (Guesses for Equations with Variables at Top-Level)**.** *Given an equation $x = t$ such that the variable x does not occur in t, the term t is both a point and gap existential guess for x:*

$$G^{\bullet}_{\exists}(K\ t,\ \lambda x.\ (x = t)) \qquad\qquad G^{-}_{\exists}(K\ t,\ \lambda x.\ (x = t))$$

Lemma 2.2.2 (Point Existential Guesses for Equations)**.** *Given an equation $t_1(x) = t_2(x)$ and a term i_c such that the formula $t_1(i_c) = t_2(i_c)$ holds, the term i_c is a point existential guess for x. This means that the following holds.*

$$t_1(i_c) = t_2(i_c) \Longrightarrow G^{\bullet}_{\exists}(K\ i_c,\ \lambda x.\ (t_1(x) = t_2(x)))$$

Therefore, unification can be used to find point existential guesses for equations.

2.2.2 Dichotomies

Many datatype definitions provide dichotomy theorems of the form

$$\forall x.\ x = c_1\ \vee\ (\exists fv.\ x = c_2(fv)) \qquad \text{and} \qquad \forall fv.\ c_1\ \neq c_2(fv)$$

The option- and list-datatypes of HOL 4, for example, provide the following theorems:

$$\begin{array}{ll} \forall x.\ x = \mathsf{NONE}\ \vee\ (\exists v.\ x = \mathsf{SOME}(v)) & \forall v.\ \mathsf{NONE} \neq \mathsf{SOME}(v) \\ \forall x.\ x = [\,]\ \vee\ (\exists hd, tl.\ x = hd :: tl) & \forall hd, tl.\ [\,] \neq hd :: tl \end{array}$$

Such theorems can be exploited for creating guesses:

Lemma 2.2.3 (Dichotomy Guesses)**.**

$$\begin{array}{lcl} (\forall fv.\ c_1 \neq c_2(fv)) & \Longrightarrow & G^{\bullet}_{\forall}(\lambda fv.\ c_2(fv),\ \lambda x.\ (x = c_1)) \\ (\forall x.\ x = c_1\ \vee\ (\exists fv.\ x = c_2(fv))) & \Longrightarrow & G^{\neg}_{\forall}(\lambda fv.\ c_2(fv),\ \lambda x.\ (x = c_1)) \end{array}$$

Exploiting dichotomy is very useful in practice. Holfoot uses it to reason about lists. The variable *pdata'* in the list-reversal example of the introduction can, for example, be instantiated using the dichotomy of lists. Without this rule, Holfoot's inference rules for singly-linked list predicates would need to unfold the data-content manually.

Many other datatypes unluckily have more than just two cases. Currently, general n-chotomies cannot be exploited, because this would require an extension of the guess concept. However, monochotomies can be handled.

2.2.3 Monochotomies

Datatypes like pairs and records do not have two cases, but only one. This structure can be exploited as well. Notice, that for this case, the guess does not depend on the predicate P any more. It just depends on the type of the variable we try to instantiate.

Lemma 2.2.4 (Monochotomy Guesses)**.**

$$\begin{array}{lcl} (\forall x.\ \exists fv.\ x = c(fv))) & \Longrightarrow & \forall P.\ G^{\neg}_{\exists}(\lambda fv.\ c(fv),\ P) \\ (\forall x.\ \exists fv.\ x = c(fv))) & \Longrightarrow & \forall P.\ G^{\neg}_{\forall}(\lambda fv.\ c(fv),\ P) \end{array}$$

2.2.4 Other guesses

The base guesses presented above are at the most important ones. However, other trivial base guesses are exploited as well for the implementation. The most important ones are point guesses that can be derived from some context information:

Lemma 2.2.5 (Context Guesses)**.**

$$\begin{array}{l} P(c) \Longrightarrow G^{\bullet}_{\exists}(K\ c,\ \lambda x.\ P(x)) \\ \neg P(c) \Longrightarrow G^{\bullet}_{\forall}(K\ c,\ \lambda x.\ P(x)) \end{array}$$

Moreover, generating guesses on the fly for constants is important to implement lifting efficiently:

Lemma 2.2.6 (Guesses for Constants)**.** *Given a term c such that a variable x does not occur in c, any term i is a valid existential and universal guess:*

$$G_{\exists}(i,\ \lambda x.\ c) \qquad G_{\forall}(i,\ \lambda x.\ c)$$

2.3 Lifting Guesses

In Sec. 2.2 the base cases of the search for guesses have been presented. It remains to discuss, how guesses are lifted over common operators.

2.3.1 Negation

Guesses are defined with negation in mind. Therefore, it is straightforward to lift guesses over negation:

Lemma 2.3.1 (Lifting Guesses over Negation)**.**

$$\begin{array}{llll} G^{\bullet}_{\exists}(i,\ \lambda x.\ P(x)) \Rightarrow G^{\bullet}_{\forall}(i,\ \lambda x.\ \neg P(x)) & G^{\bullet}_{\forall}(i,\ \lambda x.\ P(x)) \Rightarrow G^{\bullet}_{\exists}(i,\ \lambda x.\ \neg P(x)) \\ G_{\exists}(i,\ \lambda x.\ P(x)) \Rightarrow G_{\forall}(i,\ \lambda x.\ \neg P(x)) & G_{\forall}(i,\ \lambda x.\ P(x)) \Rightarrow G_{\exists}(i,\ \lambda x.\ \neg P(x)) \\ G_{\bar{\exists}}(i,\ \lambda x.\ P(x)) \Rightarrow G_{\bar{\forall}}(i,\ \lambda x.\ \neg P(x)) & G_{\bar{\forall}}(i,\ \lambda x.\ P(x)) \Rightarrow G_{\bar{\exists}}(i,\ \lambda x.\ \neg P(x)) \end{array}$$

2.3.2 Disjunction

Lifting guesses over a disjunction $P_1(x) \vee P_2(x)$ is a bit more complicated. Point existential guesses are straightforward to lift. If we have a point existential guess for either P_1 or P_2, it is also one for the disjunction:

Lemma 2.3.2 (Lifting Point Existential Guesses over Disjunction)**.**

$$\begin{array}{lll} G^{\bullet}_{\exists}(i,\ \lambda x.\ P_1(x)) \implies G^{\bullet}_{\exists}(i,\ \lambda x.\ P_1(x) \vee P_2(x)) \\ G^{\bullet}_{\exists}(i,\ \lambda x.\ P_2(x)) \implies G^{\bullet}_{\exists}(i,\ \lambda x.\ P_1(x) \vee P_2(x)) \end{array}$$

In contrast, point universal guesses have to hold for both disjuncts:

Lemma 2.3.3 (Lifting Point Universal Guesses over Disjunction)**.**

$$G^{\bullet}_{\forall}(i,\ \lambda x.\ P_1(x)) \wedge G^{\bullet}_{\forall}(i,\ \lambda x.\ P_2(x)) \implies G^{\bullet}_{\forall}(i,\ \lambda x.\ P_1(x) \vee P_2(x))$$

For gap guesses, it is the other way round:

Lemma 2.3.4 (Lifting Gap Existential Guesses over Disjunction)**.**

$$G_{\bar{\exists}}(i,\ \lambda x.\ P_1(x)) \wedge G_{\bar{\exists}}(i,\ \lambda x.\ P_2(x)) \implies G_{\bar{\exists}}(i,\ \lambda x.\ P_1(x) \vee P_2(x))$$

Lemma 2.3.5 (Lifting Gap Universal Guesses over Disjunction)**.**

$$\begin{array}{rcl} G_{\bar{\forall}}(i,\ \lambda x.\ P_1(x)) & \Longrightarrow & G_{\bar{\forall}}(i,\ \lambda x.\ P_1(x) \vee P_2(x)) \\ G_{\bar{\forall}}(i,\ \lambda x.\ P_2(x)) & \Longrightarrow & G_{\bar{\forall}}(i,\ \lambda x.\ P_1(x) \vee P_2(x)) \end{array}$$

For weak existential guesses, existential guesses for both subterms are needed. However, Lemma 2.2.6 leads to a nice rule, provided the variable does not occur in one of the operands.

Lemma 2.3.6 (Lifting Existential Guesses over Disjunction)**.**

$$\begin{array}{rcl} G_{\exists}(i,\ \lambda x.\ P_1(x)) \wedge G_{\exists}(i,\ \lambda x.\ P_2(x)) & \Longrightarrow & G_{\exists}(i,\ \lambda x.\ P_1(x) \vee P_2(x)) \\ G_{\exists}(i,\ \lambda x.\ P_1(x)) & \Longrightarrow & G_{\exists}(i,\ \lambda x.\ P_1(x) \vee p_2) \\ G_{\exists}(i,\ \lambda x.\ P_2(x)) & \Longrightarrow & G_{\exists}(i,\ \lambda x.\ p_1 \vee P_2(x)) \end{array}$$

For universal guesses it is even more complicated, because the handling of free variables does not fit well. The guess is not allowed to depend on any free variable (denoted by $K\ i_c$).

Lemma 2.3.7 (Lifting Universal Guesses over Disjunction)**.**

$$\begin{array}{rcl} \begin{array}{l} G_{\forall}(K\ i_c,\ \lambda x.\ P_1(x)) \\ G_{\forall}(K\ i_c,\ \lambda x.\ P_2(x)) \end{array} \wedge & \Longrightarrow & G_{\forall}(K\ i_c,\ \lambda x.\ P_1(x) \vee P_2(x)) \\ G_{\forall}(i,\ \lambda x.\ P_1(x)) & \Longrightarrow & G_{\forall}(i,\ \lambda x.\ P_1(x) \vee p_2) \\ G_{\forall}(i,\ \lambda x.\ P_2(x)) & \Longrightarrow & G_{\forall}(i,\ \lambda x.\ p_1 \vee P_2(x)) \end{array}$$

So, for the strong types of guesses, one of the dual guesses behaves nicely. In contrast, none of the weak guesses does. As motivated above, this is the main reason for introducing the stronger types of guesses.

2.3.3 Other Boolean Operations

We have seen above how to handle negation and disjunction. Other Boolean operations can be expressed in terms of these. Therefore, the above lemmata can also be used to lift guesses over other Boolean operations. However, for efficiency reasons the implementation in HOL 4 uses special, derived rules. Here, some of these rules are listed without further explanation:

Lemma 2.3.8 (Lifting Guesses over Conjunction)**.**

$$
\begin{array}{rcl}
G_{\exists}^{\bullet}(i,\ \lambda x.\ P_1(x)) \wedge G_{\exists}^{\bullet}(i,\ \lambda x.\ P_2(x)) & \Longrightarrow & G_{\exists}^{\bullet}(i,\ \lambda x.\ P_1(x) \wedge P_2(x)) \\
G_{\forall}^{\bullet}(i,\ \lambda x.\ P_1(x)) & \Longrightarrow & G_{\forall}^{\bullet}(i,\ \lambda x.\ P_1(x) \wedge P_2(x)) \\
G_{\forall}^{\bullet}(i,\ \lambda x.\ P_2(x)) & \Longrightarrow & G_{\forall}^{\bullet}(i,\ \lambda x.\ P_1(x) \wedge P_2(x)) \\
G_{\exists}^{-}(i,\ \lambda x.\ P_1(x)) & \Longrightarrow & G_{\exists}^{-}(i,\ \lambda x.\ P_1(x) \wedge P_2(x)) \\
G_{\exists}^{-}(i,\ \lambda x.\ P_2(x)) & \Longrightarrow & G_{\exists}^{-}(i,\ \lambda x.\ P_1(x) \wedge P_2(x)) \\
G_{\forall}^{-}(i,\ \lambda x.\ P_1(x)) \wedge G_{\forall}^{-}(i,\ \lambda x.\ P_2(x)) & \Longrightarrow & G_{\forall}^{-}(i,\ \lambda x.\ P_1(x) \wedge P_2(x)) \\
G_{\exists}(K\ i_c,\ \lambda x.\ P_1(x)) \wedge G_{\exists}(K\ i_c,\ \lambda x.\ P_2(x)) & \Longrightarrow & G_{\exists}(K\ i_c,\ \lambda x.\ P_1(x) \wedge P_2(x)) \\
G_{\exists}(i,\ \lambda x.\ P_1(x)) & \Longrightarrow & G_{\exists}(i,\ \lambda x.\ P_1(x) \wedge p_2) \\
G_{\exists}(i,\ \lambda x.\ P_2(x)) & \Longrightarrow & G_{\exists}(i,\ \lambda x.\ p_1 \wedge P_2(x)) \\
G_{\forall}(i,\ \lambda x.\ P_1(x)) \wedge G_{\forall}(i,\ \lambda x.\ P_2(x)) & \Longrightarrow & G_{\forall}(i,\ \lambda x.\ P_1(x) \wedge P_2(x)) \\
G_{\forall}(i,\ \lambda x.\ P_1(x)) & \Longrightarrow & G_{\forall}(i,\ \lambda x.\ P_1(x) \wedge p_2) \\
G_{\forall}(i,\ \lambda x.\ P_2(x)) & \Longrightarrow & G_{\forall}(i,\ \lambda x.\ p_1 \wedge P_2(x))
\end{array}
$$

Lemma 2.3.9 (Lifting Guesses over Implication)**.**

$$
\begin{array}{rcl}
G_{\exists}^{\bullet}(i,\ \lambda x.\ P_1(x)) \wedge G_{\forall}^{\bullet}(i,\ \lambda x.\ P_2(x)) & \Longrightarrow & G_{\forall}^{\bullet}(i,\ \lambda x.\ P_1(x) \Rightarrow P_2(x)) \\
G_{\forall}^{\bullet}(i,\ \lambda x.\ P_1(x)) & \Longrightarrow & G_{\exists}^{\bullet}(i,\ \lambda x.\ P_1(x) \Rightarrow P_2(x)) \\
G_{\exists}^{\bullet}(i,\ \lambda x.\ P_2(x)) & \Longrightarrow & G_{\exists}^{\bullet}(i,\ \lambda x.\ P_1(x) \Rightarrow P_2(x)) \\
G_{\exists}^{-}(i,\ \lambda x.\ P_1(x)) & \Longrightarrow & G_{\forall}^{-}(i,\ \lambda x.\ P_1(x) \Rightarrow P_2(x)) \\
G_{\forall}^{-}(i,\ \lambda x.\ P_2(x)) & \Longrightarrow & G_{\forall}^{-}(i,\ \lambda x.\ P_1(x) \Rightarrow P_2(x)) \\
G_{\forall}^{-}(i,\ \lambda x.\ P_1(x)) \wedge G_{\exists}^{-}(i,\ \lambda x.\ P_2(x)) & \Longrightarrow & G_{\exists}^{-}(i,\ \lambda x.\ P_1(x) \Rightarrow P_2(x)) \\
G_{\forall}(i,\ \lambda x.\ P_1(x)) \wedge G_{\exists}(i,\ \lambda x.\ P_2(x)) & \Longrightarrow & G_{\exists}(i,\ \lambda x.\ P_1(x) \Rightarrow P_2(x)) \\
G_{\forall}(i,\ \lambda x.\ P_1(x)) & \Longrightarrow & G_{\exists}(i,\ \lambda x.\ P_1(x) \Rightarrow p_2) \\
G_{\exists}(i,\ \lambda x.\ P_2(x)) & \Longrightarrow & G_{\exists}(i,\ \lambda x.\ p_1 \Rightarrow P_2(x)) \\
G_{\exists}(K\ i_c,\ \lambda x.\ P_1(x)) \wedge G_{\forall}(K\ i_c,\ \lambda x.\ P_2(x)) & \Longrightarrow & G_{\forall}(K\ i_c,\ \lambda x.\ P_1(x) \Rightarrow P_2(x)) \\
G_{\exists}(i,\ \lambda x.\ P_1(x)) & \Longrightarrow & G_{\forall}(i,\ \lambda x.\ P_1(x) \Rightarrow p_2) \\
G_{\forall}(i,\ \lambda x.\ P_2(x)) & \Longrightarrow & G_{\forall}(i,\ \lambda x.\ p_1 \Rightarrow P_2(x))
\end{array}
$$

2.3.4 Quantifiers

It remains to lift guesses over quantifiers. Let's first consider universal quantification. If the guess does depend on the quantified variables, it becomes a free variable of the guess:

Lemma 2.3.10 (Lifting Guesses over Universal Quantification (1))**.**

$$(\forall y.\ G^{\bullet}_{\exists}(\lambda fv.\ i(fv, y),\ \lambda x.\ P(x, y)) \Longrightarrow G^{\bullet}_{\exists}(\lambda fv'.\ i(\mathsf{FST}(fv'), \mathsf{SND}(fv')),\ \lambda x.\ \forall y.\ P(x, y))$$

$$(\forall y.\ G_{\forall}(\lambda fv.\ i(fv, y),\ \lambda x.\ P(x, y)) \Longrightarrow G_{\forall}(\lambda fv'.\ i(\mathsf{FST}(fv'), \mathsf{SND}(fv')),\ \lambda x.\ \forall y.\ P(x, y))$$

$$(\forall y.\ G_{\bar{\forall}}(\lambda fv.\ i(fv, y),\ \lambda x.\ P(x, y)) \Longrightarrow G_{\bar{\forall}}(\lambda fv'.\ i(\mathsf{FST}(fv'), \mathsf{SND}(fv')),\ \lambda x.\ \forall y.\ P(x, y))$$

$$(\forall y.\ G_{\bar{\exists}}(\lambda fv.\ i(fv, y),\ \lambda x.\ P(x, y)) \Longrightarrow G_{\bar{\exists}}(\lambda fv'.\ i(\mathsf{FST}(fv'), \mathsf{SND}(fv')),\ \lambda x.\ \forall y.\ P(x, y))$$

Notice however, that weak existential and point universal guesses can't be lifted that way. If a guess does not depend on the quantified variable, it is possible, though:

Lemma 2.3.11 (Lifting Guesses over Universal Quantification (2))**.**

$$\begin{array}{rcl}
(\forall y.\ G^{\bullet}_{\exists}(i,\ \lambda x.\ P(x, y)) & \Longrightarrow & G^{\bullet}_{\exists}(i,\ \lambda x.\ \forall y.\ P(x, y)) \\
(\forall y.\ G_{\exists}(K\ i_c,\ \lambda x.\ P(x, y)) & \Longrightarrow & G_{\exists}(K\ i_c,\ \lambda x.\ \forall y.\ P(x, y)) \\
(\forall y.\ G_{\bar{\exists}}(i,\ \lambda x.\ P(x, y)) & \Longrightarrow & G_{\bar{\exists}}(i,\ \lambda x.\ \forall y.\ P(x, y)) \\
(\forall y.\ G^{\bullet}_{\forall}(i,\ \lambda x.\ P(x, y)) & \Longrightarrow & G^{\bullet}_{\forall}(i,\ \lambda x.\ \forall y.\ P(x, y)) \\
(\forall y.\ G_{\forall}(i,\ \lambda x.\ P(x, y)) & \Longrightarrow & G_{\forall}(i,\ \lambda x.\ \forall y.\ P(x, y)) \\
(\forall y.\ G_{\bar{\forall}}(i,\ \lambda x.\ P(x, y)) & \Longrightarrow & G_{\bar{\forall}}(i,\ \lambda x.\ \forall y.\ P(x, y))
\end{array}$$

Rules for lifting guesses over existential quantification are dual. However, unique existential quantification needs a bit of attention:

Lemma 2.3.12 (Lifting Guesses over Unique Existential Quantification)**.**

$$\begin{array}{rcl}
(\forall y.\ G^{\bullet}_{\forall}(i,\ \lambda x.\ P(x, y)) & \Longrightarrow & G^{\bullet}_{\forall}(i,\ \lambda x.\ \exists! y.\ P(x, y)) \\
(\forall y.\ G_{\bar{\exists}}(i,\ \lambda x.\ P(x, y)) & \Longrightarrow & G_{\bar{\exists}}(i,\ \lambda x.\ \exists! y.\ P(x, y))
\end{array}$$

2.4 Related Techniques

Above base guesses and lifting of guesses have been presented. This section now briefly discusses other techniques that enhance guess search:

2.4.1 Rewrites

A very powerful, yet simple technique for teaching the guess search about new constructs are rewrite rules. For example, the rules above cannot generate guesses for the predicate IS_SOME. By rewriting $\mathsf{IS\_SOME}(x)$ to $\exists x'.\ x = \mathsf{SOME}(x')$, however, the rules for equations and quantification as well as the dichotomy rule for option types can be used.

The HOL 4 implementation uses rewrites to add support for predicates like IS_SOME or NULL on lists. However, adding rules like $\mathsf{append}(l_1, l_2) = [\,] \Longleftrightarrow (l_1 = [\,] \ \wedge\ l_2 = [\,])$ for list-append turned out to be beneficial in practice as well.

2.4.2 Strengthening and Weakening

Rewrite rules are very useful. However, sometimes only implications instead of equivalences are available. These can be exploited as well:

Lemma 2.4.1 (Strengthening / Weakening Guesses)**.**

$$(\forall x.\ P_1(x) \Rightarrow P_2(x)) \Longrightarrow \begin{pmatrix} \forall x.\ G^{\bullet}_{\exists}(i, P_1) & \Longrightarrow & G^{\bullet}_{\exists}(i,\ \lambda x.\ P_2) & \wedge \\ \forall x.\ G^{\sim}_{\forall}(i, P_1) & \Longrightarrow & G^{\sim}_{\forall}(i,\ \lambda x.\ P_2) & \wedge \\ \forall x.\ G^{\bullet}_{\forall}(i, P_2) & \Longrightarrow & G^{\bullet}_{\forall}(i,\ \lambda x.\ P_1) & \wedge \\ \forall x.\ G^{\sim}_{\exists}(i, P_2) & \Longrightarrow & G^{\sim}_{\exists}(i,\ \lambda x.\ P_1) & \end{pmatrix}$$

2.4.3 Minimising Variable Occurrences

As for example Lemma 2.3.6 illustrates, it is easier to find guesses for a variable if it occurs in fewer positions. Therefore, the HOL 4 implementation preprocesses the term and tries to minimise the number of occurrences. For example, $\exists x.\ (f(8{+}2) = f(x{+}2)) \wedge P(f(x+2))$ is rewritten to $\exists x.\ (f(8+2) = f(x+2)) \wedge P(f(8+2))$ by this prepossessing step. This explains, why this example can be handled, whereas apparently similar ones cannot (see Table 1.1). The implementation for minimising variable occurrences is based on simple rules like $(x = t) \wedge P(x) \iff (x = t) \wedge P(t)$ or $x \neq t \vee P(x) \iff x \neq t \vee P(t)$.

2.5 Examples

In the last few sections, the method for searching guesses has been presented on an abstract level. Let's now consider a few examples to see how this methods works in practice.

2.5.1 Example 1

Let's start with the example $\exists x.\ (\forall y.\ Q(y)\ \wedge\ (x = 7))\ \wedge\ P(x)$. This example cannot be handled by existing tools. However, the lemmata presented above allow to derive a guess for this example:

$G^{\sim}_{\exists}(K\ 7, \lambda x.\ x = 7)$	(1)	from Lemma 2.2.1
$G^{\sim}_{\exists}(K\ 7, \lambda x.\ Q(y)\ \wedge\ (x = 7))$	(2)	from (1) and Lemma 2.3.8
$G^{\sim}_{\exists}(K\ 7, \lambda x.\ \forall y.\ Q(y)\ \wedge\ (x = 7))$	(3)	from (2) and Lemma 2.3.11
$G^{\sim}_{\exists}(K\ 7, \lambda x.\ (\forall y.\ Q(y)\ \wedge\ (x = 7)) \wedge P(x))$	(4)	from (3) and Lemma 2.3.8

With Lemma 2.1.11 we can therefore simplify $\exists x.\ (\forall y.\ Q(y)\ \wedge\ (x = 7))\ \wedge\ P(x)$ to $(\forall y.\ Q(y)\ \wedge\ (7 = 7))\ \wedge\ P(7)$ and with some general infrastructure further to $(\forall y.\ Q(y)) \wedge P(7)$. Here, only the trace that succeeds is presented. The search actually produces more guesses. For example, $G^{\bullet}_{\exists}(K\ 7, \lambda x.\ x = 7)$ is derived as well, but then fails to be lifted over the conjunction.

2.5.2 Example 2

The example $\forall x.\ \mathsf{IS\_NONE}(x) \vee P(x)$ illustrates partial instantiations for datatypes.

$G_{\bar{\forall}}(\lambda x'.\ \mathsf{SOME}(x'), \lambda x.\ x = \mathsf{NONE})$	(1)	from Lemma 2.2.3
$G_{\bar{\forall}}(\lambda x'.\ \mathsf{SOME}(x'), \lambda x.\ \mathsf{IS\_NONE}(x))$	(2)	rewrite of (1)
$G_{\bar{\forall}}(\lambda x'.\ \mathsf{SOME}(x'), \lambda x.\ \mathsf{IS\_NONE}(x) \vee P(x))$	(3)	from (2) and Lemma 2.3.5

Therefore, $\forall x.\ \mathsf{IS\_NONE}(x) \vee P(x)$ can be simplified to $\forall x'.\ P(\mathsf{SOME}(x'))$.

2.5.3 Example 3

Let's consider $\exists x.\ (f(8+2) = f(x+2)) \wedge P(f(x+2))$. The guess

$$G_{\exists}^{\bullet}(K\ 8, \lambda x.\ f(8+2) = f(x+2)) \quad (1) \quad \text{from Lemma 2.2.2}$$

can be easily derived. However, then we are stuck, because we cannot derive a point existential guess for the conjunct $P(f(x+2))$. So, we weaken the guess:

$$G_{\exists}(K\ 8, \lambda x.\ f(8+2) = f(x+2)) \quad (2) \quad \text{from (1) and Lemma 2.1.10}$$

Unluckily, we are still stuck, because $P(f(x+2))$ depends on x. However, our tool can rewrite the overall term such that this dependence disappears (see Sec. 2.4.3).

$G_{\exists}(K\ 8, \lambda x.\ (f(8+2) = f(x+2)) \wedge P(f(8+2)))$	(3)	from (2), Lemma 2.3.8, 2.3.1
$G_{\exists}(K\ 8, \lambda x.\ (f(8+2) = f(x+2)) \wedge P(f(x+2)))$	(4)	from (3) and rewrite

Thus, we found a guess and can instantiate the quantifier with 8.

2.5.4 Example 4

Finally, consider $\Big((\forall x.\ P_1(x) \Rightarrow P_2(x)) \wedge P_1(2)\Big) \Longrightarrow \exists x.\ P_2(x)$. First, HOL 4's general infrastructure processes this and allows us to concentrate on $\exists x.\ P_2(x)$, while using (1) $\forall x.\ P_1(x) \Rightarrow P_2(x)$ and (2) $P_1(2)$ as lemmata. Then guesses can be derived as follows:

$G_{\exists}^{\bullet}(K\ 2, \lambda x.\ P_1(x))$	(3)	from (2), Lemma 2.2.5
$G_{\exists}^{\bullet}(K\ 2, \lambda x.\ P_2(x))$	(4)	from (1), (3), Lemma 2.4.1

Thus, we found a guess and can simplify the original goal first to $\Big((\forall x.\ P_1(x) \Rightarrow P_2(x)) \wedge P_1(2)\Big) \Longrightarrow \mathit{true}$ and then further to *true*.

3 HOL 4 implementation

3.1 User Interface

The quantifier heuristics library can be found in the sub-directory `src/quantHeuristics`. The entry point to the framework is the library `quantHeuristicsLib`.

3.1.1 Conversions

Usually the library is used for converting a term containing quantifiers to an equivalent one. For this, the following high level entry points exists:

```
QUANT_INSTANTIATE_CONV     :  quant_param list -> conv
QUANT_INST_ss              :  quant_param list -> ssfrag
QUANT_INSTANTIATE_TAC      :  quant_param list -> tactic
ASM_QUANT_INSTANTIATE_TAC  :  quant_param list -> tactic
```

All these functions get a list of *quantifier heuristic parameters* as arguments. These parameters essentially configure, which heuristics are used during the guess-search. If an empty list is provided, the tools know about the standard Boolean combinators, equations and context. `std_qp` adds support for common datatypes like pairs or lists. Quantifier heuristic parameters are explained in more detail in Section 3.2.

So, some simple usage of the quantifier heuristic library looks like:

```
> QUANT_INSTANTIATE_CONV [] ``?x. (!z. Q z /\ (x=7)) /\ P x``;
val it = |- (?x. (!z. Q z /\ (x = 7)) /\ P x) <=> (!z. Q z) /\ P 7: thm

> QUANT_INSTANTIATE_CONV [std_qp] ``!x. IS_SOME x ==> P x``
val it = |- (!x. IS_SOME x ==> P x) <=> !x_x'. P (SOME x_x'): thm
```

Usually, the quantifier heuristics library is used together with the simplifier using `QUANT_INST_ss`. Besides interleaving simplification and quantifier instantiation, this has the benefit of being able to use context information collected by the simplifier:

```
> QUANT_INSTANTIATE_CONV [] ``P m ==> ?n. P n``
Exception- UNCHANGED raised

> SIMP_CONV (bool_ss ++ QUANT_INST_ss []) [] ``P m ==> ?n. P n``
val it = |- P m ==> (?n. P n) <=> T: thm
```

It's usually best to use `QUANT_INST_ss` together with e. g. `SIMP_TAC` when using the library with tactics. However, if free variables of the goal should be instantiated, then `ASM_QUANT_INSTANTIATE_TAC` should be used:

```
P x
------------------------------------
  IS_SOME x
  : proof

> e (ASM\_QUANT_INSTANTIATE_TAC [std_qp])
P (SOME x_x') : proof
```

There is also `QUANT_INSTANTIATE_TAC` as a shortcut for using `QUANT_INSTANTIATE_CONV` as a tactic. It does not instantiate free variables or considers the assumptions.

3.1.2 Unjustified Guesses

Most heuristics justify the guesses they produce and therefore allow to prove equivalences. However, the implementation also supports unjustified guesses, which may be bogus. Let's consider the example $\exists x.\ P(x) \Longrightarrow (x = 2) \ \wedge \ Q(x)$. Because nothing is known about P and Q, we can't find a safe instantiation for x here. However, 2 looks tempting and is probably sensible in many situations. (Counterexample: $P(2)$, $\neg Q(2)$ and $\neg P(3)$ hold)

`implication_concl_qp` is a quantifier parameter that looks for valid guesses in the conclusion of an implication. Then, it assumes without justification that these guesses are probably sensible for the whole implication as well. Because these guesses might be wrong, one can either use implications or expansion theorems like $\exists x.P(x) \ \Leftrightarrow (\forall x.\ (x \neq c) \Rightarrow \neg P(x)) \Rightarrow P(c)$.

```
> QUANT_INSTANTIATE_CONV [implication_concl_qp] ``?x. P x ==> (x = 2) /\ Q x``
Exception- UNCHANGED raised

> QUANT_INSTANTIATE_CONSEQ_CONV [implication_concl_qp] CONSEQ_CONV_STRENGTHEN_direction
     ``?x. P x ==> (x = 2) /\ Q x``
val it =
   |- (P 2 ==> Q 2) ==> ?x. P x ==> (x = 2) /\ Q x: thm

> EXPAND_QUANT_INSTANTIATE_CONV [implication_concl_qp] ``?x. P x ==> (x = 2) /\ Q x``
val it = |- (?x. P x ==> (x = 2) /\ Q x) <=>
            (!x. x <> 2 ==> ~(P x ==> (x = 2) /\ Q 2)) ==> P 2 ==> Q 2: thm

> SIMP_CONV (std_ss++EXPAND_QUANT_INST_ss [implication_concl_qp]) [] ``?x. P x ==> (x = 2) /\ Q x``
val it =
   |- (?x. P x ==> (x = 2) /\ Q x) <=> (!x. x <> 2 ==> P x) ==> P 2 ==> Q 2: thm
```

The following entry points should be used to exploit unjustified guesses:

```
QUANT_INSTANTIATE_CONSEQ_CONV  :  quant_param list -> directed_conseq_conv
EXPAND_QUANT_INSTANTIATE_CONV  :  quant_param list -> conv
EXPAND_QUANT_INST_ss           :  quant_param list -> ssfrag
QUANT_INSTANTIATE_CONSEQ_TAC   :  quant_param list -> tactic
```

3.1.3 Debugging

To debug the guess-search, it is possible to print tracing information. This is done by setting the trace `QUANT_INSTANTIATE_HEURISTIC` to 1 or 2. For the example in Sec. 2.5.2 the debug output looks like:

```
val _ = set_trace "QUANT_INSTANTIATE_HEURISTIC" 1
val thm = QUANT_INSTANTIATE_CONV [std_qp] ``!x. IS_NONE x \/ P x``

searching guesses for ``x`` in ``~(IS_NONE x \/ P x)``
  searching guesses for ``x`` in ``IS_NONE x \/ P x``
    searching guesses for ``x`` in ``IS_NONE x``
      searching guesses for ``x`` in ``x = NONE``
      7 guesses found for ``x`` in ``x = NONE``
       - guess_exists_point (``NONE``, [], X)
       - guess_forall_point (``SOME x_x``, [x_x], X)
       - guess_forall (``SOME x_x``, [x_x], X)
       - guess_forall (``SOME x_x'``, [x_x'], X)
       - guess_exists (``NONE``, [], X)
       - guess_forall_gap (``SOME x_x'``, [x_x'], X)
       - guess_exists_gap (``NONE``, [], X)
    7 guesses found for ``x`` in ``IS_NONE x``
     - guess_exists_point (``NONE``, [], X)
     - guess_forall_point (``SOME x_x``, [x_x], X)
     - guess_forall (``SOME x_x``, [x_x], X)
     - guess_forall (``SOME x_x'``, [x_x'], X)
     - guess_exists (``NONE``, [], X)
     - guess_forall_gap (``SOME x_x'``, [x_x'], X)
     - guess_exists_gap (``NONE``, [], X)
    searching guesses for ``x`` in ``P x``
    no guesses found for ``x`` in ``P x``
  4 guesses found for ``x`` in ``IS_NONE x \/ P x``
   - guess_exists_point (``NONE``, [], X)
   - guess_forall (``SOME x_x'``, [x_x'], X)
   - guess_exists (``NONE``, [], X)
   - guess_forall_gap (``SOME x_x'``, [x_x'], X)
4 guesses found for ``x`` in ``~(IS_NONE x \/ P x)``
 - guess_forall_point (``NONE``, [], X)
 - guess_forall (``NONE``, [], X)
 - guess_exists (``SOME x_x'``, [x_x'], X)
 - guess_exists_gap (``SOME x_x'``, [x_x'], X)
searching guesses for ``x_x'`` in ``~P (SOME x_x')``
  searching guesses for ``x_x'`` in ``P (SOME x_x')``
  no guesses found for ``x_x'`` in ``P (SOME x_x')``
no guesses found for ``x_x'`` in ``~P (SOME x_x')``

val thm = |- (!x. IS_NONE x \/ P x) <=> !x_x'. P (SOME x_x'):  thm
```

3.1.4 Interface Details

The high level interface is mainly build around an highly configurable conversion and a corresponding consequence conversion. `EXTENSIBLE_QUANT_INSTANTIATE_CONV` gets the following arguments:

`cache_ref_opt : quant_heuristic_cache ref option` a cache for guesses. If `NONE` is passed, a new cache is created on the fly. New caches can be created by `mk_quant_heuristic_cache` and then used to cache results through multiple calls.

`re : bool` redescent into a term after some instantiation has been found? It determines, whether internally `DEPTH_CONV` or `REDEPTH_CONV` is used.

`min_var_occs : bool` add a preprocessing step to minimise the number of occurrences of a variable (see Sec.2.4.3). Since this preprocessing might be slow, one might want to skip it.

expand : bool use expansion to exploit unjustified guesses?

ctx : thm list a list of theorems that come from the context (e. g. collected by the simplifier).

base_arg : quant_param a parameter that should always be used. This is by default `basic_qp`, which can handle the standard Boolean connectives, equations and context. In special circumstances, it might be beneficial to use `empty_qp`, though.

args : quant_param list list of quantifier heuristic parameters to use.

The consequence conversion `EXTENSIBLE_QUANT_INSTANTIATE_CONSEQ_CONV` gets the same arguments, except `expand` and `ctx`. Since it can use implications, expansion is not needed. Because `DEPTH_CONSEQ_CONV` collects its own context and can't easily be used with the simplifier anyhow, `ctx` is unnecessary. `EXTENSIBLE_QUANT_INST_ss` is a wrapper of the conversion that removes the arguments `re` and `ctx`, because they are taken care of by the simplifier.

All other entry points, including the ones presented above are derived form the conversion and consequence conversion. By default the argument `cache_ref_opt` is set to `NONE`, `re` is false, preprocessing turned on, expanding turned off, an empty context is used and `basic_qp` is always present. Versions of the high level entry points containing `FAST` in their name, turn preprocessing off; `EXPAND` means that expansion is turned on and `RE` that `re` is set to true.

3.1.5 Explicit Instantiations

A special (slightly degenerated) use of the framework, is turning guess search off completely and providing instantiations explicitly. The tactic `QUANT_TAC` allows this. This means that it allows to partially instantiate quantifiers at subpositions with explicitly given terms. As such, it can be seen as a generalisation of `EXISTS_TAC`.

```
val it = !x. (!z. P x z) ==> ?a b.    Q a        b z : proof

> e( QUANT_INST_TAC [("z", `0`, []), ("a", `SUC a'`, [`a'`])] )

val it = !x. (    P x 0) ==> ?  b a'. Q (SUC a') b z : proof
```

This tactic is implemented using oracle guesses. It normally produces implications, which is fine when used as a tactic. There is also a conversion called `INST_QUANT_CONV` with the same functionality. For a conversion, implications are problematic. Therefore, the simplifier and Metis are used to prove the validity of the explicitly given instantiations. This succeeds usually only for simple examples.

3.2 Quantifier Heuristic Parameters

Quantifier heuristic parameters play a similar role for the quantifier instantiation library as simpsets do for the simplifier. They contain theorems, ML code and general configuration parameters that allow to configure guess-search. There are predefined parameters that handle common constructs and the user can define own parameters.

3.2.1 Quantifier Heuristic Parameters for Common Datatypes

There are `option_qp`, `list_qp`, `num_qp` and `sum_qp` for option types, lists, natural numbers and sum types respectively. Some examples are displayed in the following table:

$$\begin{array}{rcl}
\forall x.\ \texttt{IS_SOME}(x) \Rightarrow P(x) & \Longleftrightarrow & \forall x'.\ P(\texttt{SOME}(x')) \\
\forall x.\ \texttt{IS_NONE}(x) & \Longleftrightarrow & \mathit{false} \\
\forall l.\ l \neq [\,] \Rightarrow P(l) & \Longleftrightarrow & \forall h, l'.\ P(h :: l') \\
\forall x.\ x = c + 3 & \Longleftrightarrow & \mathit{false} \\
\forall x.\ x \neq 0 \Rightarrow P(x) & \Longleftrightarrow & \forall x'.\ P(\texttt{SUC}(x'))
\end{array}$$

3.2.2 Quantifier Heuristic Parameters for Tuples

For tuples the situation is peculiar, because each quantifier over a variable of a product type can be instantiated. The challenge is to decide which quantifiers should be instantiated and which new variable names to use for the components of the pair. There is a quantifier heuristic parameter called `pair_default_qp`. It first looks for subterms of the form $(\lambda(x_1, \ldots, x_n).\ \ldots)\ x$. If such a term is found, x is instantiated with $(x_1, \ldots, x_n)$. Otherwise, subterms of the form $\texttt{FST}(x)$ and $\texttt{SND}(x)$ are searched. If such a term is found, x is instantiated as well. This parameter therefore allows simplifications like:

$$\begin{array}{rcl}
\forall p.\ (x = \texttt{SND}(p)) \Rightarrow P(p) & \Longleftrightarrow & \forall p_1.\ P(p_1, x) \\
\exists p.\ (\lambda(p_a, p_b, p_c).P(p_a, p_b, p_c))\ p & \Longleftrightarrow & \exists p_a, p_b, p_c.\ P(p_a, p_b, p_c)
\end{array}$$

`pair_default_qp` is implemented in terms of the more general quantifier heuristic parameter `pair_qp`, which allows the user to provide a list of ML functions. These functions get the variable and the term. If they return a tuple of variables, these variables are used for the instantiation, otherwise the next function in the list is called or - if there is no function left - the variable is not instantiated. In the example of $\exists p.\ (\lambda(p_a, p_b, p_c).P(p_a, p_b, p_c))\ p$ these functions are given the variable p and the term $(\lambda(p_a, p_b, p_c).P(p_a, p_b, p_c))\ p$ and return $\texttt{SOME}(p_a, p_b, p_c)$.

3.2.3 Quantifier Heuristic Parameter for Records

Records are similar to pairs, because they can always be instantiated. Here, it is interesting that the necessary monochotomy lemma comes from HOL 4's `Type_Base` library. This means that `record_qp` is stateful. If a new record type is defined, the automatically proven monochotomy lemma is then automatically used by `record_qp`. In contrast to the pair parameter, the one for records gets only one function instead of a list of functions to decide which variables to instantiate. However, this function is simpler, because it just needs to return true or false. The names of the new variables are constructed from the field-names of the record. The quantifier heuristic parameter `default_record_qp` expands all records.

3.2.4 Stateful Quantifier Heuristic Parameters

The parameter for records is stateful, as it uses knowledge from `Type_Base`. Such information is not only useful for records but for general datatypes. The quantifier heuristic parameter `TypeBase_qp` uses automatically proven theorems about new datatypes to exploit mono- and dichotomies. Moreover, there is also a stateful `pure_stateful_qp` that allows the user to explicitly add other parameters to it. `stateful_qp` is a combination of `pure_stateful_qp` and `TypeBase_qp`.

3.2.5 Standard Quantifier Heuristic Parameter

The standard quantifier heuristic parameter `std_qp` combines the parameters for lists, options, natural numbers, the default one for pairs and the default one for records.

3.3 User defined Quantifier Heuristic Parameters

The user is also able to define own parameters. There is `empty_qp`, which does not contain any information. Several parameters can be combined using the function `combine_qps`. Together with the basic types of user defined parameters that are explained below, these functions provide an interface for user defined quantifier heuristic parameters.

3.3.1 Rewrites / Conversions

As discussed in Sec. 3.3.1, adding rewrites is a very powerful technique. `rewrite_qp` allows to provide rewrites in the form of rewrite theorems. For the example of `IS_SOME` discussed in Sec. this looks like:

```
> val thm = QUANT_INSTANTIATE_CONV [] ‘‘!x. IS_SOME x ==> P x‘‘
Exception- UNCHANGED raised

> val IS_SOME_EXISTS = prove (‘‘IS_SOME x = (?x'. x = SOME x')‘‘, Cases_on ‘x‘ THEN SIMP_TAC std_ss []);
val IS_SOME_EXISTS = |- IS_SOME x <=> ?x'. x = SOME x': thm

> val thm = QUANT_INSTANTIATE_CONV [rewrite_qp[IS_SOME_EXISTS]] ‘‘!x. IS_SOME x ==> P x‘‘
val thm = |- (!x. IS_SOME x ==> P x) <=> !x'. IS_SOME (SOME x') ==> P (SOME x'): thm
```

To clean up the result after instantiation, theorems used to rewrite the result after instantiation can be provided via `final_rewrite_qp`.

```
> val thm = QUANT_INSTANTIATE_CONV [rewrite_qp[IS_SOME_EXISTS], final_rewrite_qp[option_CLAUSES]]
    ‘‘!x. IS_SOME x ==> P x‘‘
val thm = |- (!x. IS_SOME x ==> P x) <=> !x'. P (SOME x'): thm
```

If rewrites are not enough, `conv_qp` can be used to add conversions:

```
> val thm = QUANT_INSTANTIATE_CONV [] ‘‘?x. (\y. y = 2) x‘‘
Exception- UNCHANGED raised

> val thm = QUANT_INSTANTIATE_CONV [convs_qp[BETA_CONV]] ‘‘?x. (\y. y = 2) x‘‘
val thm = |- (?x. (\y. y = 2) x) <=> T: thm
```

3.3.2 Strengthening / Weakening

In rare cases, equivalences that can be used for rewrites are unavailable. There might be just implications that can be used for strengthening or weakening (see Sec. 2.4.2). The function `imp_qp` might be used to provide such implication.

```
> val thm = QUANT_INSTANTIATE_CONV [list_qp] ``!l. 0 < LENGTH l ==> P l``
Exception- UNCHANGED raised

> val LENGTH_LESS_IMP = prove (``!l n. n < LENGTH l ==> l <> []``, Cases_on `l` THEN SIMP_TAC list_ss []);
val LENGTH_LESS_IMP = |- !l n. n < LENGTH l ==> l <> []: thm

> val thm = QUANT_INSTANTIATE_CONV [imp_qp[LENGTH_LESS_IMP], list_qp] ``!l. 0 < LENGTH l ==> P l``
val thm =
   |- (!l. 0 < LENGTH l ==> P l) <=>
      !l_t l_h. 0 < LENGTH (l_h::l_t) ==> P (l_h::l_t): thm

> val thm = SIMP_CONV (list_ss ++ QUANT_INST_ss [imp_qp[LENGTH_LESS_IMP], list_qp]) []
             ``!l. SUC (SUC n) < LENGTH l ==> P l``
val thm =
   |- (!l. SUC (SUC n) < LENGTH l ==> P l) <=>
      !l_h l_t_h l_t_t_t l_t_t_h. n < SUC (LENGTH l_t_t_t) ==> P (l_h::l_t_h::l_t_t_h::l_t_t_t): thm
```

3.3.3 Filtering

Sometimes, one might want to avoid to instantiate certain quantifiers. The function `filter_qp` allows to add ML-functions that filter the handled quantifiers. These functions are given a variable x and a term $P(x)$. The tool only tries to find instantiate x in $P(x)$, if all filter functions return *true*.

```
> val thm = QUANT_INSTANTIATE_CONV [] ``?x y z. (x = 1) /\ (y = 2) /\ (z = 3) /\ P (x, y, z)``
val thm = |- (?x y z. (x = 1) /\ (y = 2) /\ (z = 3) /\ P (x,y,z)) <=> P (1,2,3): thm

> val thm = QUANT_INSTANTIATE_CONV [filter_qp [fn v => fn t => (v = ``y:num``)]]
   ``?x y z. (x = 1) /\ (y = 2) /\ (z = 3) /\ P (x, y, z)``
val thm = |- (?x y z. (x = 1) /\ (y = 2) /\ (z = 3) /\ P (x,y,z)) <=>
              ?x   z. (x = 1) /\            (z = 3) /\ P (x,2,z): thm
```

3.3.4 Satisfying and Contradicting Instantiations

As the satisfy library[1] demonstrates, it is often useful to use unification and explicitly given theorems to find instantiations. In addition to satisfying instantiations, the quantifier heuristics framework is also able to use contradicting ones. The theorems used for finding instantiations usually come from the context. However, `instantiation_qp` allows to add additional ones:

```
> val thm = SIMP_CONV (std_ss++QUANT_INST_ss[]) [] ``P n ==> ?m:num. n <= m /\ P m``
Exception- UNCHANGED raised

> val thm = SIMP_CONV (std_ss++QUANT_INST_ss[instantiation_qp[arithmeticTheory.LESS_EQ_REFL]]) []
              ``P n ==> ?m:num. n <= m /\ P m``
> val thm =
   |- P n ==> ?m:num. n <= m /\ P m = T : thm
```

[1] see `src/simp/src/Satisfy.sml`

3.3.5 Di- and Monochotomies

As discussed in Sec. 2.2.2, dichotomies can be exploited for guess search. `distinct_qp` provides an interface to add theorems of the form $\forall x.\ c_1(x) \neq c_2(x)$. `cases_qp` expects theorems of the form $\forall x.\ (x = \exists fv.c_1(fv)) \ \vee \ldots \vee (x = \exists fv.c_n(fv))$. These theorems are for $n = 2$ used with Lemma 2.2.3 and for $n = 1$ with Lemma 2.2.4. All other cases are currently ignored.

3.3.6 Lifting Theorems

In Section 2.3 theorems have been presented that allow lifting guesses. The function `inference_qp` enables the user to provide it's own lifting inference rules. Those rules have to be theorems of the form $G_1 \wedge \ldots \wedge G_n \Longrightarrow G$, where all G_i are guesses that might be universally quantified and G is a guess. Examples for such inference rules can be found in Section 2.3. Usually, inferences look like Lemma 2.3.8. However, also rules like Lemma 2.3.10 or 2.3.11 are supported.

Example 3.3.1. When trying to add support for the Boolean operation of equivalence, there are two choices. One can use rewriting and replace every occurrence of $P_1 \Leftrightarrow P_2$ with for example $(P_1 \wedge P_2) \vee (\neg P_1 \wedge \neg P_2)$. However, this may lead to an exponential blowup of the size of the original term, since both P_1 and P_2 occur twice in the result. Therefore, it is more efficient to provide a new lifting inference for equivalences. Such a rule can easily be derived using the existing rules for basic Boolean operations.

3.3.7 Oracle Guesses

Sometimes, the user does not want to justify guesses. The tactic `QUANT_TAC` is implemented using oracle guesses for example. A simple interface to oracle guesses is provided by `oracle_qp`. It expects a ML function that given a variable and a term returns an pair of an instantiation and the free variables in this instantiation.

As an example, lets define a parameter that states that every list is non-empty:

```
val dummy_list_qp = oracle_qp (fn v => fn t =>
  let
     val (v_name, v_list_ty) = dest_var v;
     val v_ty = listSyntax.dest_list_type v_list_ty;

     val x = mk_var (v_name ^ "_hd", v_ty);
     val xs = mk_var (v_name ^ "_tl", v_list_ty);
     val x_xs = listSyntax.mk_cons (x, xs)
  in
     SOME (x_xs, [x, xs])
  end)
```

Notice, that an option type is returned and that the function is allowed to throw `HOL_ERR` exceptions. With this definition, the call

```
NORE_QUANT_INSTANTIATE_CONSEQ_CONV [dummy_list_qp]
   CONSEQ_CONV_STRENGTHEN_direction ``?x:'a list y:'b. P (x, y)``
```

results in `(?y x_hd x_tl. P (x_hd::x_tl,y)) ==> ?x y. P (x,y) : thm`.

3.3.8 User defined Quantifier Heuristics

At the lowest level, our tool searches guesses using ML-functions called *quantifier heuristics*. Slightly simplified, such a quantifier heuristic gets a variable and a term and returns a set of guesses for this variable and term. Heuristics allow full flexibility. However, to write your own heuristics a lot of knowledge about the ML-datastructures and auxiliary functions is required. Therefore, no details are discussed here. Please have a look at the source code and contact Thomas Tuerk (`tt291@cl.cam.ac.uk`), if you have questions. `heuristics_qp` and `top_heuristics_qp` provide interfaces to add user defined heuristics to a quantifier heuristics parameter.

4 Conclusion

The quantifier heuristic is a powerful tool to instantiate quantifiers. It subsumes the power of the existing unwind and satisfy libraries. More importantly though, it is very flexible and easily extendable by the user. Moreover, it can be used to instantiate quantifiers using unjustified guesses.