Department of Mathematics and Computer Science
Institute of Computer Science

Software Model Checking With Higher-Order Automated Theorem Provers: A Logic Embedding Approach

Maximilian Claus
m.claus@fu-berlin.de

Submitted in partial fulfilment of the requirements for the degree of
Master of Science in Computer Science

Submission date: 24.11.2015
Supervisor: PD Dr. Christoph Benzmüller
Second examiner: Prof. Dr. Raúl Rojas

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, den _____ _____

Maximilian Claus

Abstract

Research in automating classical higher-order logic (HOL) has been on the rise for several years now and as a result, automated theorem provers for HOL, such as Satallax or LEO-II, are readily available. The area of application for these new tools is vast: The expressivity of the logic makes it easy to translate deduction problems of many facades into equivalent problems in higher-order logic. This work is particularly concerned with solving problems in software verification, which can be addressed by, among others, the technique of *model checking*. In model checking, the state space of a program is inspected and exhaustively tested to whether it fulfils a certain property of interest or not. Such a property is commonly expressed in a verification logic, mostly temporal logics like LTL or CTL. We show how both the state space inspection and the verification logic can be embedded in HOL, and therefore how problems in model checking can be converted to problems for HOL theorem provers. The result is a verification software which is not optimised for one specific verification logic, but can interpret program properties in an arbitrary logic, given an appropriate embedding in HOL. Furthermore, we present improvements for the current automated provers, which are required to treat such generated problems efficiently. These additions might also be highly relevant for problems in different domains and therefore thoroughly discussed.

Contents

1 Introduction

In software engineering, model checking is an umbrella term for various techniques of automated verification. In an abstract sense, most verification problems can be stated as model checking problems, where the program is a putative model of some logical formula. However, nowadays the term is used in a somewhat narrower sense: Programs amenable to model-checking are usually finite-state and do not operate on arbitrary input data, which can potentially be infinite. Decision procedures are thus decidable and their runtime is foreseeable. Model checking therefore sharply contrasts formal systems for the verification of algorithms, such as Hoare logic [17], which make correctness statements for programs receiving arbitrary inputs and whose decision procedures are necessarily undecidable. It is unknown whether Hoare logic can be successfully automated: Most algorithms' correctness proofs rely on the induction principle, whose automation alone poses a significant challenge, and also require guessing these inductive properties in the first-place, which might be even more difficult.

Because it is a decidable problem, model checking presents huge potential for the development of highly optimised decision procedures, which can prove or disprove various properties about programs. But its lesser algorithmic complexity makes it also interesting in other ways: Whilst interactive proof assistants are commonly used to verify complex programs manually in Hoare logic or similar formal systems, the aim of this thesis is to use fully automated provers for software verification. Naturally, model checking is a good start for such a venture.

Motivation This thesis is motivated by the discovery that theorem provers for classical higher-order logic can be successfully employed to reason about several non-classical logics through the technique of *semantic embeddings*. Examples of such semantic embeddings and their applications include [6], [10], [7].

Model checking is perhaps the most notable practical application of modal logics today, and it might thus come as a natural question to what degree theorem provers are suitable for the automated verification of software through this technique. To this end, a translation tool is developed which takes programs (or models) and correctness properties and translates these to putative theorems in HOL. Furthermore, a semantic embedding of the popular verification logic of *Linear Temporal Logic* is given, and the software is tested with this embedding on a small number of sample programs.[1]

[1] The source code of the software and associated files are available via GitHub on https://github.com/MaximilianC/HOLMCF. Access rights may be requested via e-mail to the author.

Outline This thesis is structured as follows: Section 2 explains the technique of explicit-state model checking, as used in the remainder of the thesis. This section is based on the respective book chapters of "Principles of Model Checking" by Baier & Katoen [4]. However, most of the formalisms have been altered or changed altogether with regard to formalisation in higher-order logic. Section 3 introduces syntax and semantics of higher-order logic. It also discusses the representation of some basic data structures in higher-order logic and points to weaknesses of its most basic incarnation *simply type theory*. Section 4 presents the inner workings of HOLMCF in detail and goes through the formalisation of all necessary concepts, especially those presented more informally in section 2. Section 5 motivates a refinement for normalising higher-order logic formulae and subsequently describes an implementation of a new normalisation strategy including these refinements. Finally, section 6 presents various tests conducted with HOLMCF and points to future work.

2 Basic Model Checking

Model checking is a technique for the verification of both hardware and software. In fact, no constraints need to be posed on the type of computational object to be investigated, but in practice these turn out to be mostly imperative programs or sequential circuits. In either case, we simply refer to them as *programs*. Formally, we formulate a correctness property ϕ in a specification logic, whose models happen to be programs, and want to determine if our program M is a model for ϕ.

$$M \vDash \phi$$

Clearly, via *Rice's theorem*, the problem of inferring non-trivial properties about arbitrary programs is undecidable. In order to regain decidability, most model checkers restrict the type of program they operate on in one way or the other. In most cases, the input programs need to have only finitely many states (*finite-state model checking*). With this restriction in mind, the specific strength of most model checkers lies in the verification of reactive systems. Reactive systems are programs which interact with their environment over time (e.g. webservers, autopilots, surveillance systems etc.) in contrast to terminating programs, which realise mathematical functions (e.g. compilers, model checkers themselves or any kind of algorithm in the classical sense). The primary source of complexity in the design of reactive systems is concurrency, which usually is a requirement posed by the particular environment the system is designed for. In contrast, concurrency for terminating programs is mostly used to improve performance.

In the presence of concurrency appears the *state space explosion problem*: The number of states in a program can grow exponentially in the number of execution threads (or *processes*). For humans, these magnitudes quickly become intractable for even the smallest of programs. Model checkers address this problem by running a brute force search over the whole state space of the program and checking the property in question individually for each state. Needless to say, the algorithmic cost of such a procedure is remarkably high, and different techniques have been developed to improve search strategies. Commonly, one differentiates between two techniques:

- Explicit-state model checking
- Symbolic model checking

Explicit-state model checkers generate all possible states of a program, as they would occur during run-time. Symbolic model checkers on the other hand represent the set of states more indirectly through some compressed data structure or set of logical formulae to save memory. Symbolic model checking was first introduced by McMillan in his thesis [20]. Historically, the

first symbolic model checkers used *reduced ordered binary decision diagrams* (ROBDDs). In reality however, explicit-state model checkers use heuristics to trim down the number of generated states likewise, so the distinction might not always be clear. In this thesis, we are only interested in explicit-state model checking and keep the search procedure as simple as possible.

2.1 Linear Temporal Logic

2.1.1 Intuition

When investigating an imperative program M, it is not obvious in which language one should state the putative correctness property ϕ. At first sight, it might be sufficient to only prove sentences of the kind "For all states $s \in M$, $P(s)$ holds.", where P is a sentence in classical propositional logic. On second thought, we might also be inclined to claim things like "For all $s \in M$, if $P(s)$ holds, $Q(s')$ holds for some later state s'." Imagine for instance a formula which asserts the correctness of a program that, upon the push of a button, is supposed to print something on the screen. It becomes clear that a suitable language should be some sort of logic (it talks about truth), and that it should incorporate some notion of time. It should thus be a *temporal logic*. The first proposition for such a logic in computer science is due to Amir Pnueli and is called *linear temporal logic* [22].

Linear temporal logic (LTL) is a modal logic with modalities $\{\Box, \Diamond, \bigcirc\}$ called *always*, *eventually*, and *next* respectively as well as two additional connectives $\{\mathcal{U}, \mathcal{R}\}$ called *until* and *release*. With these new constructs at hand, our previous examples can be stated more concisely as $\Box P$ ("P always holds.") and $\Box(P \Rightarrow \Diamond Q)$ ("It is always the case that if P holds, Q will hold at some later point."). Additionally, one can make statements about the immediate future: $\Box(P \Rightarrow \bigcirc Q)$ says that whenever P holds, Q has to hold in the next instant (time is discrete!). As usual with truth-functional logics, only very few connectives are required to obtain the expressivity of a complete signature. In this case, operators $\{\bigcirc, \mathcal{U}\}$ already suffice to express the remainder.

2.1.2 Syntax & Semantics

Unlike classical modal logic, which is interpreted over possible worlds of an accessibility relation (a Kripke structure), LTL is interpreted over *sequences of worlds* of such a relation. In such sequences, the i-th world denotes the "state of affairs" at time i. Sequences of this kind are also called *words* or `paths`.

For our purposes, the set of possible worlds W consists of sets of propositions over the state space domain of a program. If the domain of the state space consists only of tuples of numerical values, it should be sufficient to have equality and relational operators $(=, <, >)$ at hand to build propositions. If one enriches the underlying programming language to also support arrays and similar complex structures, the means of building propositions should be extended to be able to construct more interesting predicates. These predicates could for instance check whether an array is sorted or whether it only contains even numbers and so on. Verifying such expressive properties (and formalising a respective language) goes beyond the scope of this thesis and we restrict ourselves to (finite) integer variables, thus having only tuples of integers as states. Furthermore, for demonstration, we here restrict ourselves to forming propositions through equality. Other means of forming propositions, like relational operators, can be added in a straightforward manner.

As an example, assume a program with two variables a and b with the binary domain $\{0, 1\}$. The resulting set of possible worlds W then is the symmetric closure of $\{\{a = 0\}, \{a = 1\}, \{b = 0\}, \{b = 1\}, \{a = 0, b = 0\}, \{a = 1, b = 0\}, \{a = 0, b = 1\}, \{a = 1, b = 1\}\}$.

After having agreed what type of propositions are of interest to us, we can define the syntax of the logic formally.

Definition 1 Linear Temporal Logic Syntax
Let V be the set of variables of an underlying program and $v \in V$. Let N be the domain of the variables (for instance integers) and $n \in N$. The set of well-formed LTL formulae $\mathcal{WFF}$ is generated by the following grammar:

$$tv ::= v \mid n$$
$$f, g ::= (tv = tv) \mid \neg f \mid f \wedge g \mid \bigcirc f \mid \Diamond f \mid \Box f \mid f \,\mathcal{U}\, g$$

Expressions formed by the rule tv (which are simply either variable names or constant values) are *time-dependent* values. The only means of building atomic propositions is to compare these time-dependent values to each other, which means in effect variables to other variables or variables to constant values.

We continue with the semantics of LTL formulae: Formally, we can interpret sequences as functions from the natural numbers to worlds. As in classical modal logic, we give a Kripke-style semantics to formulas by defining a satisfaction relation $\models \subseteq (\mathbb{N} \to W) \times \mathcal{WWF}$ where $\mathbb{N} \to W$ is the set of sequences.

Let $\sigma \in \mathbb{N} \to W$.

$$
\begin{array}{ll}
\sigma \models \text{true} & \\
\sigma \models tv_1 = tv_2 & \text{iff } (tv_1 = tv_2) \in \sigma(0) \\
\sigma \models \phi \wedge \psi & \text{iff } \sigma \models \phi \text{ and } \sigma \models \psi \\
\sigma \models \neg\phi & \text{iff } \sigma \not\models \phi \\
\sigma \models \bigcirc\phi & \text{iff } \sigma' \models \phi \text{ where } \sigma'(i) = \sigma(i+1) \\
\sigma \models \Diamond\phi & \text{iff } \exists j.\ \sigma' \models \phi \text{ where } \sigma'(i) = \sigma(i+j) \\
\sigma \models \Box\phi & \text{iff } \forall j.\ \sigma' \models \phi \text{ where } \sigma'(i) = \sigma(i+j) \\
\sigma \models \phi\, \mathcal{U}\, \psi & \text{iff } \exists j\ . \forall k < j.\ \sigma' \models \phi \text{ and } \sigma'' \models \psi \\
 & \quad \text{where } \sigma'(i) = \sigma(i+k) \text{ and } \sigma''(i) = \sigma(i+j)
\end{array}
$$

Figure 1: Semantics of LTL given by its satisfaction relation

With the satisfaction relation at hand, we can define what it means for a formula to be valid. Intuitively, a formula is valid if and only if it is satisfied by all paths of the Kripke structure. We introduce an inductively defined predicate $\text{isPath}_{K,s}(\sigma)$, which is true for all functions $\sigma \in \mathbb{N} \to W$ which are indeed paths in Kripke structure K with initial world s. It is reasonable to assume some definite initial world s, the same way we can assume a definite initial state for every program we run on a computer.

Definition 2 Linear Temporal Logic Validity
A formula ϕ in linear temporal logic is valid for a Kripke structure K and initial world $s \in W$ if and only if

$$\forall\sigma.\ \text{isPath}_{K,s}(\sigma) \Rightarrow \sigma \models \phi$$

where $\text{isPath}_{K,s}(\sigma)$ is defined as:

$$\text{isPath}_{K,s}(\sigma) \equiv (\sigma(0) = s) \wedge (\forall i.\ (\sigma(i), \sigma(i+1)) \in K)$$

2.2 State Space Generation

While the state space generation for single threaded programs is easy, it is a bit more involved for concurrent programs. It has to be noted that whenever we use the term "state space" we are not talking about merely a set of states, but a graph structure that also captures the relationship between these states. This requirement gives rise to the idea of a *transition system*.

We follow the approach by Baier & Katoen and transform an imperative program into an intermediate form called a program graph, which makes

the concurrency more explicit. Generating the transition system from a program graph is then straightforward. Figure 1 illustrates this process for a very small example program.

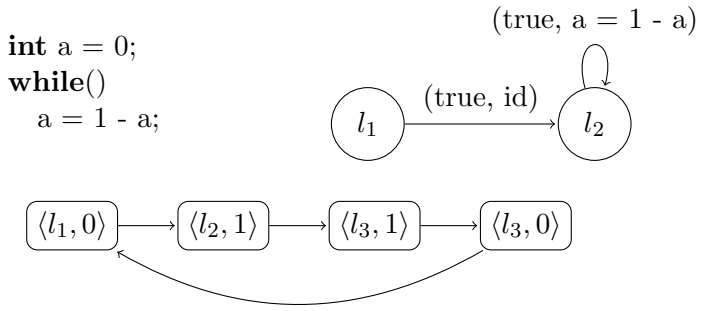


Figure 2: Program, program graph and transition system (below)

2.2.1 Imperative Programs

To begin with we need to define the structure of programs.

Definition 3 Imperative Programs Syntax
Let V be a set of variable names and $v \in V$. A program s is a word generated by the following grammar:

$$c ::= \text{true} \mid \text{false} \mid c_1 || c_2 \mid c_1 \&\& c_2 \mid !c \mid e_1 == e_2 \mid e_1 < e_2 \mid e_1 > e_2$$
$$e ::= v \mid e_1 + e_2 \mid e_1 - e_2$$
$$s ::= \textbf{skip} \mid v = e \mid \textbf{if } c \textbf{ then } s_1 \textbf{ else } s_2 \mid \textbf{while}(c)\ s \mid s_1\ ;\ s_2 \mid (s_1 | s_2)$$

It might be worth noticing that the above grammar is ambiguous and thus not suitable for parsing. We accept this shortcoming as the price to pay for brevity. The actual grammar used in this thesis is more complex and essentially a subset of the C programming language. It is included as part of the appendix. Whenever necessary any arising ambiguity will be resolved by appropriate means.

With the program syntax in place it is now important to address semantics. We assume a standard semantics as it is common for this type of language. Thus we only need to clarify the precise meaning of the parallel composition operator |. This is usually done through the *interleaving model*, where a concurrent program is understood as a *set of sequential programs*. When running the concurrent program, essentially one of these sequential programs is run, but we do not know which one!

The main consideration when using the interleaving model is deciding on the amount of *granularity*. The set of sequential programs for a concurrent program is determined by all possible arrangements of the atomic instructions into which a parallel program is decomposed. It must therefore be decided on how small an atomic instruction must be. Baier & Katoen propose two different levels of granularity: *test-and-set* semantics and *two-step* semantics. In the former, atomic instructions are of shape "**if** X**then**Y;", such that condition X is tested and Y possibly executed in one step. In the latter, atomic instructions are either conditions or actions, such that after testing X and before executing Y, an interleaving might occur.

Neither model is representative of actual hardware: In both of them, conditions can be of arbitrary complexity, for instance, checking an arbitrary amount of variables for some value.

In this thesis, we chose the *two-step* semantics, because they come closer to the behaviour of real hardware and also being the semantics chosen by the popular *Spin* [18] model checker.

We give an operational semantics for programs through a mapping from programs to *program graphs*, whose construction is described in detail in the next section.

2.2.2 Program Graphs

A program graph is an intermediate structure used in the process of generating the state space. It consists of a graph whose edges are labeled with two functions, a *guard* and an *action*. The edges are also called *transitions*. The nodes are labeled with *locations*, which roughly denote the current state of execution of the program. They can be thought of as states of the program counter register on physical machines.

A program graph may be "executed" as follows: For a given transition, the guard function receives the current memory configuration (the contents of the memory) as an input and determines whether the respective transition is *enabled*. For all enabled transitions, one might pick this transition by applying the *action* function to the current memory configuration to receive a new memory configuration and continue this process for the transitions of the next node. The circumstance that more than one transition might be active at given time reflects the nondeterministic nature of concurrent programs.

It is important to note that in purely sequential programs, there is always at most one enabled transition! No enabled transition at all means the program is in a terminal state.

In foresight of modeling program graphs in higher-order logic, we make an extra effort in defining the set of locations very precisely.

Definition 4 Locations
The set of locations is the smallest set of terms generated by the following uninterpreted functions:

- $\overline{\text{start}} \in Loc$
- $\overline{\text{end}} \in Loc$
- $\overline{\text{if}} : Loc \to Loc$
- $\overline{\text{else}} : Loc \to Loc$
- $\overline{\text{while}} : Loc \to Loc$
- $\overline{\text{seq}} : Loc \to Loc$
- $\overline{\text{par}} : Loc \to Loc$
- $\langle , \rangle^2 : Loc \to Loc \to Loc$

As an equivalence relation on locations we assume the usual congruence closure.

In most scenarios such a complicated construction for representing locations is not necessary. Instead one might simply have a counter which is incremented whenever a fresh location is needed, such that each location is represented uniquely by some number. In our setting, formalising program graphs in higher-order logic, the opposite is true: Formalising natural numbers and relevant operations is at least as involved as the construction above. Furthermore, some additional state has to be passed around when building the program graph to ensure that all used numbers are unique, which is explicitly avoided through this approach.

Definition 5 Program Graph
A program graph

$$PG : Loc \times (Mem \to Bool) \times (Mem \to Mem) \times Loc$$

is a set[3] of quadruples with the following meaning:

- Loc is the set of *locations*,
- Mem is the set of possible memory configurations.

[2]This function may be called "choice"

[3]Commonly graphs are defined as a pair of nodes and edges. It is sufficient for our purposes to only look at the edges of the graph and thus to avoid some redundancy.

- $Mem \to Bool$ is the *guard* function
- $Mem \to Mem$ is the *action* function.

The construction function $C_{PG}(P, s)$ for a program P with inital location s maps to a pair of type $(Loc \to PG) \times Loc$. The first element of this pair is a function which yields a program graph if it applied to some location, which then shall be the final location of the resulting graph (that is, the location with no outgoing transitions). The second element is such a potential final location. Setting up the construction function like this allows for the construction of cycles, as it is for instance required in the rule of *while*-loops. For instance, when building a graph $(PG, f) = C_{PG}(P, s)$, which is supposed to loop back to its initial location, one can simply discard the returned location f and apply s again to PG. $PG(s)$ is then a program graph which has a cycle from s to s.

Program graphs are defined recursively over the structure of programs:

skip	$C_{PG}(\textbf{skip}, s_1) =$ $\quad (\lambda s_2.\ \{(s_1, True, id, s_2)\},\ \overline{\text{seq}}\ s_1)$
$x = v$	$C_{PG}(x = v, s_1) =$ $\quad (\lambda s_2.\ \{(s_1, True, update\ x\ v, s_2)\},\ \overline{\text{seq}}\ s_2)$
if(p) $\quad e_1$ **else** $\quad e_2$	$C_{PG}(\textbf{if}\ p\ \textbf{then}\ e_1\ \textbf{else}\ e_2, s_1) =$ $\quad (\lambda s_2.\ \{(s_1, p, id, \overline{\text{if}}\ s_1), (s_1, \neg p, id, \overline{\text{else}}\ s_1)\}$ $\qquad \cup\ Q_1(s_2) \cup Q_2(s_2),\ \overline{\text{seq}}\ s)$ where $(Q1, \_) = C_{PG}(e_1, \overline{\text{if}}\ s_1)$ $\qquad (Q2, \_) = C_{PG}(e_2, \overline{\text{else}}\ s_1)$
while(p) $\quad e$	$C_{PG}(\textbf{while}\ p\ e, s_1) =$ $\quad (\lambda s_2.\ \{(s_1, p, id, \overline{\text{while}}\ s_1), (s_1, \neg p, id, s_2)\} \cup Q(s_1),\ \overline{\text{seq}}\ s)$ where $(Q, \_) = C_{PG}(e, \overline{\text{while}}\ s_1)$
$e_1;\ e_2$	$C_{PG}(e_1; e_2, s_1) =$ $\quad (\lambda s_4.\ Q(s_2) \cup Q(s_4), s_3)$ where $(Q1, s_2) = C_{PG}(e_1, s_1)$ $\qquad (Q2, s_3) = C_{PG}(e_2, s_2)$

$e_1 \mid e_2$	$C_{PG}(e_1 \mid e_2, s_1) =$ $\quad (\lambda s_2.\ R(s_2)[\langle s_1, s_1\rangle \leftarrow s_1, \langle s_2, s_2\rangle \leftarrow s_2],\ \overline{\text{par}}\ s_1)$ where $(Q_1, \_) = C_{PG}(e_1, s_1)$ $(Q_2, \_) = C_{PG}(e_2, s_1)$ $R(t) = \{(\langle s_1', s_1\rangle, a, b, \langle s_2', s_1\rangle) \mid (s_1', a, b, s_2') \in Q_1(t)\}$ $\cup\{(\langle s_1, s_1'\rangle, a, b, \langle s_1, s_2'\rangle) \mid (s_1', a, b, s_2') \in Q_2(t)\}$ $\cup\{(\langle s_1, s\rangle, a, b, \langle s_2, s\rangle) \mid (s_1, a, b, s_2) \in Q_1(t), (\_, \_, \_, s) \in Q_2(t)\}$ $\cup\{(\langle s, s_1\rangle, a, b, \langle s, s_2\rangle) \mid (s_1, a, b, s_2) \in Q_2(t), (\_, \_, \_, s) \in Q_1(t)\}$

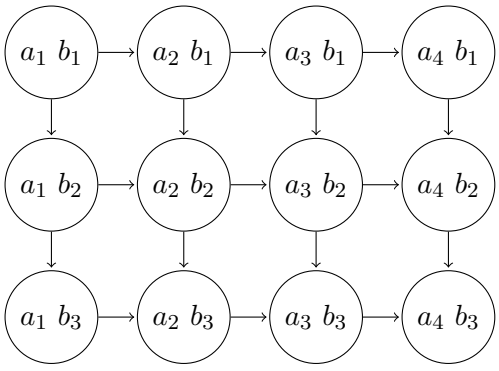
Table 1: Program Graph Construction Rules

Note that in the rule for parallel construction, the substitutions in R shall denote that locations $\langle s_1, s_1\rangle$ and $\langle s_2, s_2\rangle$ are to be replaced by initial location s_1 and final location s_2 respectively to connect these new edges to the remaining graph.

The parallel composition rule is more complicated than the other rules and requires additional explanation. Imagine the interleaving of the following simple sequences:

$$\{\ a_1;\ a_2;\ a_3;\ a_4;\ \} \mid \{\ b_1\ ;b_2\ ;b_3;\ \}$$

Some possible executions for this interleaving would for instance be $a_1\ b_1\ a_2\ a_3\ b_2\ a_4\ b_3$ or $b_1\ a_1\ a_2\ a_3\ b_2\ b_3\ a_4$. All these combinations are executions resulting from the following grid-shaped program graph:



Every edge in the graph represents the execution of one statement, the other execution thread stays the same. No matter how the graph is traversed, by the time the bottom right vertex is reached, all statements have been executed.

Given two other program graphs, the parallel composition rule will arrange their nodes in such a grid to make the parallelism explicit. With this structure it is then easy to extract the state space out of a program graph by an iterative process called *unfolding*.

2.2.3 Unfolding Program Graphs Into Transition Systems

A transition system (see figure 2) is a graph representing the state space of the program and the relationship between these states. Thus the set of vertices of the graph is the set of reachable states in the program, which are connected by a directed edge if a state can be reached from the other. States are determined uniquely by some location (which represents the program counter), and a memory configuration. They are therefore elements of $Loc \times Mem$. Building the transition system lies at the heart of the model checking process: To see whether a program models an LTL formula ϕ, we check if all infinite *paths* over its transition system satisfy ϕ.

We take the same approach as with program graphs and represent the graph only by the relation of its vertices (i.e. states). We may do this because all distinct states must be connected, unconnected states can not turn up in a transition system (unless there is only one state and in this case there exist no infinite paths).

A transition system $TS(PG, m_0)$ for a program graph PG and an initial memory configuration m_0 can be defined inductively: We know it must be the case that $(\overline{\text{start}}\ , m_0, m, l) \in TS(PG, m_0)$ for all successor states (m, l) of $(\overline{\text{start}}\ , m_0)$, and we know that if $(m, l, m', l') \in TS(PG, m_0)$, so must be $(m', l', m'', l'') \in TS(PG, m_0)$ for all successor states (m'', l'') of (m', l') and (m, l) being arbitrary. We can also capture this notion of *successor states* formally:

Definition 6 Successor States
The set of successor states $Succ((l, m))$ of a state $(l, m) \in Loc \times Mem$ in a program graph PG is defined as

$$\{(l', m') \mid \exists (l, g, a, l') \in PG \,.\, g(m) \wedge a(m) = m'\}$$

Formalising a transition system is now easy:

Definition 7 Transition System
Given a program graph PG and an initial memory configuration m_0, a transition system

$$TS(PG, m_0) : (Loc \times Mem) \times (Loc \times Mem)$$

is a set of quadruples built to abide the following law:

$$(\forall (l, m) \in Succ((\overline{\text{start}}, m_0)).\ (\overline{\text{start}}, m_0, l, m) \in TS) \wedge$$
$$(\forall (\_, \_, l, m) \in TS.\ \forall (l', m') \in Succ(l, m).\ (l, m, l', m') \in TS)$$

In practice however, we may wish to "compute" the transition system explicitly, which can be done iteratively: We observe that, starting out with the empty set, we can successively add transitions to newly discovered states. This process must eventually come to a halt because the number of states a program can have is limited (remember we are doing *finite-state* model checking). This finiteness is enforced by the finite memory representation (Mem is a finite set).

$$\text{enabled\_trans}(l, m) = \{(l_1, g, a, l_2) \mid (l_1, g, a, l_2) \in PG \wedge l_1 = l \wedge a\ m\}$$

$$\text{UNFOLD}(PG, TS, active) =$$
$$\quad \text{disc\_trans} := \bigcup_{(\_,\_,l,m) \in active} \{(l_1, m, l_2, f\ m) \mid (l_1, g, f, l_2) \in \text{enabled\_trans(l, m)}\}$$
$$\quad \text{new\_trans} := \{(l_1, m_1, l_2, m_2) \mid (l_1, m_1, l_2, m_2) \in \text{disc\_trans} \wedge (l_1, m_1, l_2, m_2) \notin TS\}$$
$$\quad \textbf{if}\ \text{new\_trans} = \emptyset$$
$$\qquad \textbf{then}\ \text{new\_trans} \cup TS$$
$$\qquad \textbf{else}\ \text{UNFOLD}(PG, \text{new\_trans} \cup TS, \text{new\_trans})$$

$$TS(PG, m_0) = \text{UNFOLD}(PG, \emptyset, \{(\_, \_, \overline{\text{start}}, m_0)\})$$

Figure 3: Transition system unfolding function

The above algorithm keeps two different sets TS and *active*. The former contains the transition system built so far, and the latter the set of active states (in a purely sequential program only one state could be active at a time). Note that we use underscores to signalise that values are either unused or their value is arbitrary. From the set of active states, the enabled transitions in the program graph are determined (enabled_trans). A program graph transition is enabled iff its location matches the one of the active state and its guard function evaluates to true under the memory configuration of the state. From these enabled transitions, new transitions in the transition system can be found (these are called disc_trans for "discovered transitions"). Some of these newly discovered transitions may have been found before, so we filter those out which are already contained in the set TS. If no new transitions are to be found anymore, we can terminate the search.

3 Higher-Order Logic And Automated Theorem Proving

3.1 Simple Type Theory

The term "higher-order logic" is used for extensions of first-order logic which also permit quantification over relations (in addition to the domain of individual atoms). Church's *Simple Type Theory* (STT) is one of such extensions and unarguably one of the more popular ones in computer science (in its different incarnations). STT is essentially a simply typed lambda calculus enriched with a set of special constant symbols serving as the logical signature. The set of types T is generated from a set of base types (at least including the Boolean type o) and the binary type constructor $\to$ for function spaces. The set of well-formed formulae $\mathcal{WWF}$ is defined inductively as follows (assuming $s, t \in \mathcal{WWF}, \alpha \in T$):

$$\begin{aligned} s, t ::= {} & p_\alpha \mid X_\alpha \mid (\lambda X_\alpha . s_\beta)_{\alpha\to\beta} \mid (s_{\alpha\to\beta}\ t_\alpha)_\beta \mid (\neg_{o\to o} s_\sigma)_o \\ & \mid (s_o \wedge_{o\to o\to o} t_o)_o \mid (s_\alpha =_{\alpha\to\alpha\to o} t_\alpha)_o \mid (\Pi_{(\alpha\to o)\to o}\ s_{\alpha\to o})_o \end{aligned}$$

The $\Pi_{(\alpha\to o)\to o}$ modifier is used for universally quantifying the first argument of the function it is passed. We often rewrite its applications to functions with binders $\Pi_{(\alpha\to o)\to o}(\lambda X_\alpha.\ s_o)$ as explicit quantifications $(\forall X_\alpha.\ s_o)$. We also use other logical connectives such as $\vee, \Rightarrow$ and the existential quantifier $\exists$ as usual (the original notation for existential quantification is $\Sigma_{(\alpha\to o)\to o}$ in analogy to $\Pi_{(\alpha\to o)\to o}$). Finally, we employ *let*-syntax on the term level occasionally to introduce local definitions: The construct **let** $x_1 = e_1; x_2 = e_2; ...; x_n = e_n$ **in** y translates to $(\lambda x_1.\ (\lambda x_2.\ ...\ (\lambda x_n.\ y)\ e_n)\ e_2)\ e_1$.

It is worthwhile to note the discovery by Andrews [1], that primitive equality itself is in fact a complete signature: It is sufficient to express quantification, it holds that $\forall X_\alpha . s_o \equiv (\lambda X_\alpha . s_0) = (\lambda X_\alpha . \top)$, as well as conjunction: it holds that $X_o \wedge Y_o \equiv \lambda X\ Y.\ \lambda f.\ f\ X\ Y = f\ \top\ \top)$, where $\top$ may be defined as $(=_{o\to o\to o}) = (=_{o\to o\to o})$.

From here onwards the term HOL is used interchangeably with STT when talking about higher-order logic in general.

3.2 Semantics

Under its standard interpretation, STT is incomplete due to Gödel's famous incompleteness theorem (one can easily formulate the Peano axioms to define the natural numbers). In this case STT is strictly stronger a logic than first-order logic. There is however an alternative meaningful semantics due

to Henkin [16] which restores completeness, but necessarily reduces STT's expressive strength to effectively first-order logic.

Although formulations of HOL have enjoyed a widespread use in interactive proof systems dating back to the 1960's with the advent of Robin Milner's LCF system, it is only recently that there has been an increased interest in its semantics among computer scientists. This is mostly due to desire of automating HOL. An automatic proof system can barely be of any use if its underlying calculus is incomplete and there is no indication to whatever class of theorems it may actually be able to prove!

As a result of the novelty of automatic provers, there is only very few information on Henkin semantics in the computer science literature. The underlying idea is as follows:

The standard semantics of higher-order logic is set-theoretic: When quantifying over some predicate $P : A \to o$, we know the set of possible instances to be isomorphic to $\mathcal{P}(A)$. Higher-order logic thus allows for strict control over the cardinality of models. In first-order logic, we know by the Löwenheim-Skolem theorem that such a distinction is impossible. Even more so, it is the case that if a first-order formula has a model, it also has Herbrand model, which is countable by definition. Prove methods for first-order logic rely on this countability to find appropriate substitution instances in the Herbrand universe, whether by enumeration, unification, or some other method.

Henkin models share this property of countability: Instead of interpreting function spaces set-theoretically, the domains of the quantifiers are defined through comprehension axioms as elements of the term-language (which is the simply typed lambda calculus). This again allows for an exhaustive search of substitution instances during proof search.

3.3 Automation

Development of powerful automated reasoners for HOL is still at an early stage. Nevertheless, there already exists a small number of automated tools to chose from:

Satallax is an automated prover for simply type theory by Chad E. Brown [12]. It uses a tableau calculus whose derivations map to propositional clauses. A SAT-solver is used to periodically check the generated clauses for unsatisfiability.

LEO-II is an automated prover developed by Benzmüller et al. [9] based on the resolution method. Its successor LEO-III is currently in development.

TPS is the oldest automated prover for HOL, written by Andrews et. al. Its interactive counterpart is ETPS, which is used for educational purposes. [2]

Nitpick is a counter-example finder for HOL developed by Jasmin Blanchette [11]. It is part of the Isabelle/HOL system and predominantly used to find mistakes when devising formalisations.

Sledgehammer, a part of the Isabelle/HOL system, contains a translation tool from HOL to first-order logic, which can be exploited to apply the wide range of mature first-order provers to higher-order problems. Strong candidates for attempts to prove such translated formulae are the Vampire [19] and E [24] systems.

3.4 Semantic Embeddings

3.4.1 HOL as a Universal Logic

Higher-order logic, while not more powerful than first-order logic, is a practical host language for semantic embeddings of other logics. By "semantic embedding" we refer to a technique in which the truth of a formula is entirely determined through the model theory of its logic. This is in contrast to proof-theoretic embeddings, where inference rules of a logical calculus are encoded in the host-logic and the truth of a formula is determined by the existence of a proof object (i.e. a sequence of valid inferences). The approach of semantic embeddings has been successfully applied before to reason with and about various modal logics with automated higher-order provers. [10] [7]

3.4.2 Embedding Linear Temporal Logic

We restate the semantics of the LTL connectives again as an embedding in HOL. To do so, we employ the well-known *Church numerals* to encode natural numbers over an otherwise uninterpreted base type N. Because we intend to use the embedding for model checking, we already refine the set of possible worlds to a set of program states: States are encoded as mappings from variables (elements of type Sym) to integers (elements of type Int), the requirement of adding some sort of program counter (for which we have defined the set of *locations*) is discussed later on. Of course, the defined variables may vary with the input program we are reasoning about. To make the embedding more suggestive, we introduce several type synonyms, signalised by the keyword **type**. For instance, for the former two constructs

we introduce type synonyms Nat and $State$ respectively. It follows naturally that paths are functions from natural numbers to states.

The essential idea of the semantic embedding is contained in the type for LTL formulae. These are mappings from paths to truth values. To check whether a path models a formula, we simply apply the path to it. It follows that a formula is valid if it is modeled by all paths.

We refrain from embedding the $\mathcal{U}$-connective because it is more difficult to embed than the other connectives and we do not really need it. For a direct translation from the previously stated semantics, subtraction of Church numerals is required, which is messy, and an additional quantifier needs to be introduced. It is however possible to get rid of the additional quantifier because it is bounded and it may be unfolded into a sequence of conjunctions on the fly. This could perhaps be done through a more specific representation of Church numerals, such that one can apply a formula to a Church numeral to generate such a sequence. It is unclear whether this is a good, or even possible solution.

basetype N

type $Nat = (N \to N) \to (N \to N)$
type $State = Sym \to Int$
type $Path = Nat \to State$
type $TV = State \to Int$
type $Form = Path \to o$

zero : $Nat = \lambda f\ x.\ x$
succ : $Nat \to Nat = \lambda n.\ \lambda f\ x.\ f\ (n\ f\ x)$
add : $Nat \to Nat \to Nat = \lambda m\ n.\ \lambda f\ x.\ m\ f\ (n\ f\ x)$

$^C : Int \to TV$	$\equiv \lambda i.\ \lambda s.\ i$
$^V : Sym \to TV$	$\equiv \lambda v.\ \lambda s.\ s\ v$
$=^{LTL}: TV \to TV \to Form$	$\equiv \lambda v_1 v_2.\ \lambda p.\ v_1\ (p\ \text{zero}) = v_2\ (p\ \text{zero})$
$\text{true}^{LTL} : Form$	$\equiv \lambda p.\ \text{true}$
$\wedge^{LTL} : Form \to Form \to Form$	$\equiv \lambda \phi\ \psi.\ \lambda p.\ \phi\ p \wedge \psi\ p$
$\neg^{LTL} : Form \to Form$	$\equiv \lambda \phi.\ \lambda p.\ \neg(f\ p)$
$\bigcirc : Form \to Form$	$\equiv \lambda \phi.\ \lambda p.\ \phi\ (\lambda i.\ p\ (\text{succ}\ i))$
$\Diamond : Form \to Form$	$\equiv \lambda \phi.\ \lambda p.\ \exists i.\ \phi\ (\lambda j.\ p\ (\text{add}\ i\ j))$
$\Box : Form \to Form$	$\equiv \lambda \phi.\ \lambda p.\ \forall i.\ \phi\ (\lambda j.\ p\ (\text{add}\ i\ j))$

Figure 4: Semantic Embedding of LTL in HOL

Functors ^{C} and ^{V} are used to introduce constants and variables as time-dependent values. These can further be used for comparison by application to $=^{LTL}$. Essentially, a time-dependent value is a function which gets passed the current state and yields some constant value of type Int. Lifted constants of this kind ignore the current state and always return the same value, lifted variables act as selection functions and chose from the current state the value the variable is referencing.

3.5 Set Theory in HOL

3.5.1 Sets of Tuples And Set Operations

Sets are, next to functions, the most ubiquitous objects in modern mathematics. Most of the introduced constructions in this thesis so far are effectively sets. When working with sets in HOL, it is common to fall back on the Church's encoding of sets as characteristic functions. In this encoding, we identify a set S of type ι as a function $S_c : \iota \to o$, such that $s \in S \iff S_c\ s = \top$. Operations on sets can then be expressed naturally:

$$\cup := \lambda s_1 s_2.\ \lambda x.\ s_1\ x \vee s_2\ x \qquad \cap := \lambda s_1 s_2.\ \lambda x.\ s_1\ x \wedge s_2\ x$$

This is a very elegant representation when reasoning in set theory in HOL, but hopelessly inadequate for computation. Namely, to compute program graphs and transition systems, we require typical operations from the functional programming world like *map*, *fold* or *filter*. Furthermore, we do not only require sets of primitive objects but in fact sets of tuples. This requirement is more worrisome than initially obvious: For tuples we might assume that we can simply use the Curch encoding for pairs (or an n-ary variation), but unfortunately, these are not typeable without higher-rank polymorphic types![4] We see this from the typing of Church pairs in System F, a lambda calculus with arbitrary-rank polymorphic types:

$$\begin{array}{ll} (t_1, t_2) & := \Lambda\gamma.\ \lambda f_{A\to B\to\gamma}.\ f\ t_1\ t_2 \\ fst & := (\lambda p_{A\times B}.\ p\ A\ (\lambda x_A.\ \lambda y_B.\ x))_{A\times B\to A} \\ snd & := (\lambda p_{A\times B}.\ p\ B\ (\lambda x_A.\ \lambda y_B.\ y))_{A\times B\to B} \end{array}$$

Figure 5: Pairs in System F

There is no way to remove the type quantifier for the type variable γ. Instead, we might attempt to simply use appropriate axiomatisations as our

[4] However, in the special case of having tuples of values from the same set, i.e. $A \times A \times ... \times A$, Church tuples are typeable in STT.

way out, though. We could introduce a new type $\gamma_{A\times B}$ which represents the cartesian product $A \times B$. We then add appropriate equations to axiomatise pairing and projection.

basetype $\gamma_{A\times B}$

$pair : A \to B \to \gamma_{A\times B}$
$fst : \gamma_{A\times B} \to A$
$snd : \gamma_{A\times B} \to B$

axiom $fst\ (pair\ a\ b) = a$
axiom $snd\ (pair\ a\ b) = b$

Figure 6: Axiomatisation of Pairs in Simple Type Theory

Unfortunately, the current provers separate the computational aspect of HOL, term normalisation, from the reasoning part, where axioms are applied to the conjecture. This in turn results in enormous terms containing huge numbers of untreated parings and projections, such as $fst\ (pair\ a\ [...])$ instead of a. Additionally, it is not guaranteed that the axioms are exclusively applied in a sensible manner. For instance, the prover might guess that it requires some pairs somewhere and just create many useless pairs using the *pair*-axiom. In conclusion, this approach is not feasible.

Luckily, we can circumvent representing tuples directly altogether, but instead take advantage of currying. In our case:

$$A \times B \to o \simeq A \to (B \to o)$$

The type for pairs of sets is isomorphic to characteristic set funtions with an extra argument. We can therefore proceed to identify a set of pairs $S : A \times B$ with a chracteristic function $S_{c_2} : A \to (B \to o)$, such that $(a, b) \in S \iff S_{c_2}\ a\ b = \top$. For every arity n of tuples we need, we create appropriate instances of the set operations which take n arguments for the tuple object they represent as an input.

Next, we address the issue of computing with (finite) sets. What we want is to compute set comprehensions, for instance:

$$\{x \mid y \in Y \wedge x = f(y) \wedge P(y)\}$$

In functional programming terminology, we might reformulate such a comprehension by use of higher-order functions:

$$X = \text{filter}\ P\ (\text{map}\ f\ Y)$$

How can one realise such functions? We have ruled out axiomatisations for *filter* and *map*, what remains is to modify the representation of sets itself. We do so by adding an extra argument to the characteristic function, which will represent an arbitrary mapping, as follows:

$$\{1, 2, 3\} \equiv \lambda f.\ f\ 1 \vee f\ 2 \vee f\ 3$$

To test membership of some value s in some set S, one passes the comparison function $(\lambda x.\ x = s)$ to S. Passing different functions than the comparison function yields other set operations.

We will now put our two findings together, and list some set functions for sets of pairs and their higher-order functions:

type Set2 $a\ b = (a \to b \to o) \to o$

empty2 : Set2 $a\ b$
empty2 $= \lambda f.\ false$

singleton2 : $a \to b \to$ Set2 $a\ b$
singleton2 $= \lambda a\ b.\ \lambda f.\ f\ a\ b$

member2 : $a \to b \to$ Set2 $a\ b \to o$
member2 $= \lambda x_1\ x_2\ s.\ s\ (\lambda ab.a = x_1 \wedge b = x_2)$

union2 : Set2 $a\ b \to$ Set2 $a\ b \to$ Set2 $a\ b$
union2 $= \lambda s_1 s_2.\ \lambda f.\ s_1\ f \vee s_2\ f$

map2_2 : $(a \to b \to (c \to d \to o) \to o) \to$ Set2 $a\ b \to$ Set2 $c\ d$
map2_2 $= \lambda g\ s.\ \lambda f.\ s\ (\lambda z_1\ z_2.\ g\ z_1\ z_2\ f)$

filter2 : $(a \to b \to o) \to$ Set2 $a\ b \to$ Set2 $a\ b$
filter2 $= \lambda p\ s.\ \lambda f.\ s\ (\lambda x_1\ x_2.\ p\ x_1\ x_2 \wedge f\ x_1\ x_2)$

flatten2 : Set (Set2 $a\ b$) $\to$ Set2 $a\ b$
flatten2 $= \lambda s.\ \lambda g.\ s\ (\lambda x.\ x\ g)$

mapCond2 : $(a \to b \to o) \to (a \to b \to (c \to d \to o) \to o) \to$
$(a \to b \to (c \to d \to o) \to o) \to$ Set2 $c\ d \to$ Set2 $c\ d$
mapCond2 $= \lambda p\ f_1\ f_2\ s.$
$\lambda f.\ s\ (\lambda x_1\ x_2.\ (p\ x_1\ x_2 \wedge f_1\ x_1\ x_2\ f) \vee (\neg(p\ x_1\ x_2) \wedge f_2\ x_1\ x_2\ f))$

Figure 7: Set functions for sets of type $A \times B$

We suffix set functions by the arity of the tuples they contain, *empty3* would denote the empty set of triples. Notice that for function *map* we also have to take into account the result type: Function *map2_2* maps sets of pairs to sets of pairs, *map2_3* maps sets of pairs to sets of triples, etc. Function

mapCond realises a conditional mapping. Its first parameter is a predicate p which, when true for some element, applies the first function f_1 to that element and f_2 otherwise. It is necessary to manually define such a function because STT does not have a native conditional construct.

3.5.2 Discussion

We went through some lengths to model sets and operations on them on the term-level. The result is not particularly elegant, nor necessarily efficient in practice. Indeed, automated provers need to ensure the application of some simplification rules during the normalisation stage to handle these constructs well, as we will see later. These shortcomings are mostly due to the restricted type system of Simple Type Theory as used in the automated provers (no polymorphism, only function spaces).

Interactive provers solve the arising issues with extensions to the type system: Most incarnations of interactive HOL systems come with built-in types for products and disjoint unions (or even inductive data types), which makes the term language remarkably more expressive. Introducing higher-rank types (or even full System F) as a base language would result in a similar gain in expressivity.

However, in automated theorem proving, the introduction of polymorphism introduces additional complexity and requires further research. Benzmüller et al. give an example where instances of type variables need to be guessed [5]. It is not clear whether such strong support for type polymorphism is really necessary, or if instead the quantification of type variables could be carefully reduced to not hinder the proof search, while still augmenting the expressivity of the term language.

4 HOLMCF

4.1 Program Description

4.1.1 Overview

HOLMCF is a translation tool to do software model checking with automated theorem provers. When given an imperative program and a putative correctness property, it generates a logical formula in THF0 syntax [8], which is a theorem if and only if the program satisfies the property in question. It does so through formalisations of the various concepts described in the Basic Model Checking sections: program graphs, transition systems and most notably semantic embeddings of temporal logics.

The utilisation of semantic embeddings is responsible for the most remarkable feature of HOLMCF: It is parametric in the verification logic to chose! One simply has to provide an embedding to the program, passed as a text-file with necessary definitions and appropriate "wrapper" functions. The price to pay for this convenience is undoubtedly high: Even with the most efficient automated HOL reasoners at hand it will never be efficient enough to verify real-world models. The use-cases for such a system are therefore of a different nature: It can serve as a prototyping tool to design verification logics and as a reference implementation from which one can derive more efficient decision procedures.

The reason to chose HOL, despite the superior maturity of first-order provers, is twofold:

Expressivity The essential idea of model checking is the interpretation of program state spaces as Kripke-structures. Kripke semantics are heavily based on set theory and usually stated in predicate logic. It can be seen form the presented embedding of LTL that embeddings of this kind are remarkably straightforward and come "almost for free". If it would be necessary to conceive a non-trivial embedding in first-order logic, this would completely undermine the usefulness of the software in the first place.

Computation It is possible to adequately axiomatise the transition system of a given program and rely on the prover to deduce that the program is a valid model of the specification formula merely from these axioms. We can not expect any automatic proof system to deal with problems of this magnitude. Instead, we operate on transition systems (i.e. Kripke structures) directly by shifting their generation to the term-level. We rely on a prover's normalisation capabilities to reduce a program to the λ-term which denotes its transition system relation

and subsequently let the prover's inference engine reason about it. It is only through the heavy use of term-level computation that we can guide the proof systems enough to make solving these problems an actual possibility.

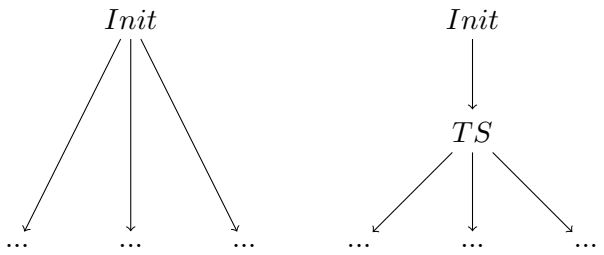


Figure 8: Blind search versus prover guidance by term-level computation of the transition system

Unfortunately, as it was already hinted to in the discussion of computing with sets in HOL (see section 3), the provers' current normalisation strategies are not sufficient to deal with the generated terms. Therefore HOLMCF features a flag to carry out the required normalisation itself, as a workaround. These issues are discussed in section 5.

HOLMCF expects two input files in two distinct languages. Programs (or models) are written in *WhilePar*, a simple imperative language with a C-like syntax.

Feature-wise it resembles the WHILE language initially presented by Hoare, which is used frequently in the study of imperative programming [17]. The *Par*-suffix emphasises that it supports non-sequential composition and that it is therefore a parallel language. Files for embeddings are stated in a custom HOL-language. Properties of models are to be found in the program file as well and share the same language with the embeddings.

It is unfortunate that HOLMCF only interprets inputs in custom languages, but the choice of existing ones is limited. A potential candidate for programs would be *Promela*, which is the language used by the *Spin* [18] model checker. Unfortunately it is based on Dijkstra's very original guarded command language, which is not very intuitive. Promela's feature set is also significantly too rich for the investigations conducted in this thesis. THF0 would be a potential candidate for writing HOL formulae. However, some regrettable trade-offs had to be made to keep it easily parsable by Prolog (the logic programming language), an important design goal for the whole TPTP language family, which renders it an unsuitable language for humans to read and write.

The program is written in Haskell and uses the lexer generator *Alex* and

parser generator *Happy* for parsing its inputs.

4.1.2 Usage

HOLMCF is invoked from the command-line with the input files' names (programs have ending ".whp" and embeddings ".emb" respectively). As mentioned, one additional flag "-n" is supported to inform HOLMCF that it shall β-normalise the formula itself, effectively already computing the complete transition system. Its use is strongly advised for all but the most trivial problems. The generated file can then be passed to a chosen HOL-ATP to do the actual verification. Satallax is currently the strongest prover and therefore a natural choice.

```
$ holmcf -n peterson.whp ltl.emb -o problem.p
$ satallax problem.p
Finished reading thf file problem.p
Initialized minisat
% SZS status Theorem
```

Using HOLMCF with Satallax from the command-line

First-order provers can be employed through the Sledgehammer translation tool provided in Isabelle/HOL, which can also be invoked from the command line (here E is used as a prover):

```
$ holmcf -n peterson.whp ltl.emb -o problem.p
$ isabelle tptp_translate FOF problem.p
        | grep -v "Exception\|\*\*\*" > problem_fof.p
$ eprover --auto-schedule --tstp-format -s --memory-limit=1024
        --cpu-limit=1000 problem_fof.p
$ satallax problem.p
[...]
# Proof found!
# SZS status Theorem
```

Using HOLMCF with first-order prover E

4.2 Implementation

4.2.1 Programs and Program Graphs

Due to the lack of inductive data types, programming with lambda calculus is noticeably different from common functional programming. To define imperative programs, the natural approach to take in a functional programming language is to model the abstract syntax tree of a program as an inductive data type. To generate a program graph, one would then subsequently

write a recursive function which builds up such a graph from a syntax tree. With no data types at our disposal, we instead define a signature of "constructor functions" of program graphs for each language construct. The same applies to arithmetic and boolean expressions used in assignments and conditions for *while*- and *if*-statements, where we build "constructor functions" for functions mapping to *action* functions (memory configurations to memory configurations) and *guard* functions (from memory configurations to truth values). Fortunately, because the elements of memory configurations are all of the same type, we can fall back on the Church encoding for tuples this time: A memory configuration of a program with three variables is a lambda term of shape $[\lambda s.\ s\ x_1\ x_2\ x_3]_{(Int\to Int\to Int\to Int)\to Int}$. For each variable i we introduce a selector function var_i to "read" from a memory configuration and an update function $updateVar_i$ to "write" to memory, i.e. create a new memory configuration. An update function has two arguments: An arithmetic expression, an element of type $NatExp$, which is a function from memory configurations to integers, and an old memory configuration.

$$\textbf{type}\ \text{Sym} = Int \to Int \to Int \to Int$$

$$\textbf{type}\ \text{Mem} = Sym \to Int$$

$$\text{var1}_{Sym} \equiv \lambda v_1\ v_2\ v_3.\ v_1$$
$$\text{var2}_{Sym} \equiv \lambda v_1\ v_2\ v_3.\ v_2$$
$$\text{var3}_{Sym} \equiv \lambda v_1\ v_2\ v_3.\ v_3$$

$$\text{updateVar1}_{NatExp\to Mem\to Mem} \equiv \lambda e\ s.(\lambda v.\ v\ (e\ s)\ (s\ \text{var2})\ (s\ \text{var3}))$$
$$\text{updateVar2}_{NatExp\to Mem\to Mem} \equiv \lambda e\ s.(\lambda v.\ v\ (s\ \text{var1})\ (e\ s)\ (s\ \text{var3}))$$
$$\text{updateVar3}_{NatExp\to Mem\to Mem} \equiv \lambda e\ s.(\lambda v.\ v\ (s\ \text{var1})\ (s\ \text{var2})\ (e\ s))$$

Figure 9: Memory selector- and update functions for programs with three variables

The above definitions are dependent on the number of variables in the input program and therefore generated by HOLMCF on the fly for a given program.

With variables in place we can proceed with the definition of conditional and arithmetic expressions. For the conditional (Boolean) operations, we can simply use the built-in logical connectives of HOL. For the arithmetic expressions, some more effort is required. Most programming languages' arithmetic operations map directly to those which can be carried out by the processor's arithmetic logic unit (ALU). Numbers are therefore stored as binary vectors of size n in two's complement representation. Today's processors often support registers for a choice of n up to 64 (64 bits), but for explicit-state model checking there is no point in letting variables range

over such huge intervals. Even enumerating all possible states of a variable of size $n = 32$ requires 16 gigabytes of memory! While $n = 8$ seems like a good choice, we will be easy on the proof systems and pick $n = 4$, thus letting variables range from -8 to 7.

For the arithmetic operations, it is equally reasonable to take their respective digital circuits and model these using HOL's in-built logical connectives. We do this exemplarily for the addition operation, which is obtained by cascading four *full adders* in series. A full adder is a circuit which adds two bits and a carry together, outputting their sum and the next carry.

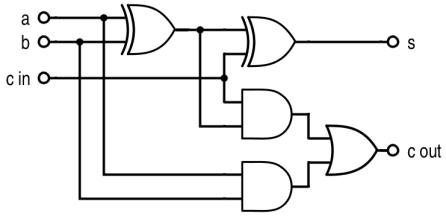


Figure 10: A full adder circuit

Such a full adder can be translated straightforwardly to a λ-term. We circumvent the problem of having to return the pair $(a + b, c_{out})$ through passing a continuation function instead, which is then invoked with both return values.

$$
\begin{aligned}
&add1_{o\to o\to o\to(o\to o\to\alpha)\to\alpha} = \\
&\quad \lambda a\ b\ c_{in}\ cont. \\
&\qquad \textbf{let}\ g1 = (a \vee b) \wedge \neg(a \wedge b) \\
&\qquad\quad\ \ g2 = (g1 \vee c_{in}) \wedge \neg(g1 \wedge c_{in}) \\
&\qquad\quad\ \ g3 = g1 \wedge c_{in} \\
&\qquad\quad\ \ g4 = a \wedge b \\
&\qquad\quad\ \ c_{out} = g3 \vee g4 \\
&\qquad \textbf{in}\ cont\ c_{out}\ g2
\end{aligned}
$$

The use of the continuation argument *cont* becomes clear when cascading the full adders to create a 4-bit adder. The initial input carry is zero ($\bot$) and the last output carry is discarded.

$$\textbf{type}\ Int = (o \to o \to o \to o \to o) \to o$$

$$n0_{o\to o\to o\to o\to o} = \lambda a\ b\ c\ d.\ d$$

$$
\begin{aligned}
&n1_{o\to o\to o\to o\to o} = \lambda a\ b\ c\ d.\ c\\
&n2_{o\to o\to o\to o\to o} = \lambda a\ b\ c\ d.\ b\\
&n3_{o\to o\to o\to o\to o} = \lambda a\ b\ c\ d.\ a
\end{aligned}
$$

$$
\begin{aligned}
&add_{Int\to Int\to Int} =\\
&\quad \lambda a\ b.\\
&\qquad add1\ (a\ n0)\ (b\ n0)\ \bot\ (\lambda c0\ r0.\\
&\qquad add1\ (a\ n1)\ (b\ n1)\ c0\ (\lambda c1\ r1.\\
&\qquad add1\ (a\ n2)\ (b\ n2)\ c1\ (\lambda c2\ r2.\\
&\qquad add1\ (a\ n3)\ (b\ n3)\ c2\ (\lambda c3\ r3.\\
&\qquad\quad (\lambda f.\ f\ r3\ r2\ r1\ r0)))))
\end{aligned}
$$

Terms $n3, n2, n1, n0$ function as bit selectors. Subtraction can be obtained by the usual technique of using 1 ($\top$) as the initial input carry and inverting all the bits of b, the second input value.

Relational operators are modeled in a similar manner. To check if $a < b$ for two numbers a and b, their bits are compared pair-wise: For each bit a_i, if all more significant bit pairs (a_j, b_j) with $j > i$ are equal, b_i may only be set if a_i is set as well. Otherwise, $a \geq b$.

$$
\begin{aligned}
&lt_{Int\to Int\to o} =\\
&\quad \lambda a\ b.\\
&\qquad a\ (\lambda a3\ a2\ a1\ a0.\\
&\qquad b\ (\lambda b3\ b2\ b1\ b0.\\
&\qquad \textbf{let}\ x3 = (a3 = b3)\\
&\qquad\quad x2 = (a2 = b2)\\
&\qquad\quad x1 = (a1 = b1)\\
&\qquad \textbf{in}\ (\neg a3 \wedge b3) \vee (x3 \wedge \neg a2 \wedge b2)\\
&\qquad\quad \vee (x3 \wedge x2 \wedge \neg a1 \wedge b1) \vee (x3 \wedge x2 \wedge x1 \wedge \neg a0 \wedge b0)))
\end{aligned}
$$

Again, the opposite operation of $a > b$ can be obtained by only a slight modification: Instead of inverting a in the final formula, b must be inverted.

These operations can now be used to model the denotational semantics of arithmetic and Boolean expressions. Such expressions, which in addition to constants may conclude program variables, are in effect mappings from memory configurations to integers and truth values.

type NatExp $= Mem \to Int$

$\text{Var}_{Sym \to NatExp} = \lambda v.\ \lambda s.\ s\ v$
$\text{NLit}_{Int \to NatExp} = \lambda n.\ \lambda s.\ n$
$\text{AddOp}_{NatExp \to NatExp \to NatExp} = \lambda e1\ e2.\ \lambda s.\ add\ (e1\ s)\ (e2\ s)$
$\text{SubOp}_{NatExp \to NatExp \to NatExp} = \lambda e1\ e2.\ \lambda s.\ sub\ (e1\ s)\ (e2\ s)$

type BoolExp $= Mem \to o$

$\text{BLit}_{o \to BoolExp} = \lambda b.\ \lambda s.\ b$
$\text{Not}_{BoolExp \to BoolExp} = \lambda e.\ \lambda s.\ \neg(e\ s)$
$\text{AndOp}_{BoolExp \to BoolExp \to BoolExp} = \lambda e1\ e2.\ \lambda s.\ e1\ s \wedge e2\ s$
$\text{EqOp}_{NatExp \to NatExp \to BoolExp} = \lambda e1\ e2.\ \lambda s.\ e1\ s = e2\ s$
$\text{LtOp}_{NatExp \to NatExp \to BoolExp} = \lambda e1\ e2.\ \lambda s.\ lt\ (e1\ s)\ (e2\ s)$
$\text{GtOp}_{NatExp \to NatExp \to BoolExp} = \lambda e1\ e2.\ \lambda s.\ gt\ (e1\ s)\ (e2\ s)$

Figure 11: Constructor functions for arithmetic- and conditional expressions

The operational behaviour of these definitions can be shown very well by means of a simple example. Assuming we have a program with two variables a and b, we would like to evaluate the assignment `a = a + (b - 1)` as an *action* function:

(updateVar1 (AddOp (Var var_1) (SubOp (Var var_2) (NLit one))))
$\to_\beta$ ($\lambda s.\ \lambda v.\ v$ (add (s ($\lambda v_1\ v_2.\ v_1$)) (sub (s ($\lambda v_1\ v_2.\ v_2$)) one)) (s var_2))

The final remaining step is the definition of the program graph itself. We remember that a program graph is a set of quadruples $Loc \times (Mem \to Bool) \times (Mem \to Mem) \times Loc$ and that locations are composed of uninterpreted function symbols. Unfortunately the introduction of these function symbols alone is not sufficient, but we have to explicitly state the uniqueness of each location through a number of distinctiveness lemmata.

$Start \neq End$	$\forall X.\text{End} \neq \text{Seq}\ X$	$\forall XY.\text{Else}\ X \neq \text{Par}\ Y$
$\forall X.Start \neq If\ X$	$\forall X.\text{End} \neq \text{Par}\ X$	$\forall X\ Y\ Z.\text{Else}\ X \neq \text{Choice}\ Y\ Z$
$\forall X.Start \neq Else\ X$	$\forall X\ Y.\text{End} \neq \text{Choice}\ X\ Y$	$\forall X\ Y.\text{While}\ X \neq \text{Seq}\ Y$
$\forall X.Start \neq While\ X$	$\forall X\ Y.\text{If}\ X \neq \text{Else}\ Y$	$\forall X\ Y.\text{While}\ X \neq \text{Par}\ Y$
$\forall X.Start \neq Seq\ X$	$\forall X\ Y.\text{If}\ X \neq \text{While}\ Y$	$\forall X\ Y\ Z.\text{While}\ X \neq \text{Choice}\ Y\ Z$
$\forall X.Start \neq Par\ X$	$\forall X\ Y.\text{If}\ X \neq \text{Seq}\ Y$	$\forall X\ Y.\text{Seq}\ X \neq \text{Par}\ Y$
$\forall X\ Y.Start \neq Choice\ X\ Y$	$\forall X\ Y.\text{If}\ X \neq \text{Par}\ Y$	$\forall X\ Y\ Z.\text{Seq}\ X \neq \text{Choice}\ Y\ Z$
$\forall X.End \neq If\ X$	$\forall X\ Y\ Z.\text{If}\ X \neq \text{Choice}\ Y\ Z$	$\forall X\ Y\ Z.\text{Par}\ X \neq \text{Choice}\ Y\ Z$
$\forall X.End \neq Else\ X$	$\forall X\ Y.\text{Else}\ X \neq \text{While}\ Y$	
$\forall X.End \neq While\ X$	$\forall X\ Y.\text{Else}\ X \neq \text{Seq}\ Y$	

Figure 12: Distinctiveness lemmata

Proof systems do not handle these distinctiveness lemmata very well. This problem will be addressed in section 5.

The construction rules for program graphs were given back in section 2. Every language construct has one such rule, each of which maps to a constructor function. Again, to circumvent the use of pairs, continuations are used. When being passed an initial location, instead of returning a value of type $(Loc \to PG) \times Loc$, a function $(Loc \to PG) \to Loc \to PG$ is passed in addition. This function is then called with the actual return values and its return value is passed through as the final return value. When a program graph is created from a program p, the expression $(p\ Start\ (\lambda c\ s.\ c\ End))$ is reduced with initial and final locations $Start$ and End.

type $PG = Set4\ Loc\ BoolExp\ (S \to S)\ Loc$

type $Prog = Loc \to ((Loc \to PG) \to Loc \to PG) \to PG$

$\text{skip}_{Prog} = \lambda s\ c.\ c\ (\lambda t.\ \text{singleton5}\ s\ (\text{BLit true})\ id\ t)\ (Seq\ s)$

$\text{assign}_{NatExp \to (NatExp \to S \to S) \to Prog} =$
  $\lambda e\ updateVar.\ \lambda s\ c.\ c\ (\lambda t.\ \text{singleton5}\ s\ (\text{BLit true})\ (updateVar\ e)\ t)\ (Seq\ s)$

$\text{ifthenelse}_{BoolExp \to Prog \to Prog \to Prog} =$
  $\lambda p\ e1\ e2.\ \lambda s\ c.$
    $e1\ (If\ s)\ (\lambda q1\ s1.$
    $e2\ (Else\ s)\ (\lambda q2\ s2.$
      $c(\lambda t.\ \text{union5}\ (\text{singleton5}\ s\ p\ id\ (If\ s))$
          $(\text{union5}\ (\text{singleton5}\ s\ (Not\ p)\ id\ (Else\ s))$
          $(\text{union5}\ (q1\ t)\ (q2\ t))))\ (Seq\ s)))$

$\text{while}_{BoolExp \to Prog \to Prog} =$
  $\lambda p\ e.\ \lambda s\ c.$
    $e\ (While\ s)\ (\lambda qs1.$
      $c\ (\lambda t.\ \text{union5}\ (\text{singleton5}\ s\ p\ id\ (While\ s))$
          $(\text{union5}\ (\text{singleton5}\ s\ (Not\ p)\ id\ t)\ (q\ s)))\ (Seq\ s))$

$\text{seq}_{Prog \to Prog \to Prog} =$
  $\lambda e1\ e2.\ \lambda s\ c.$
    $e1\ s\ (\lambda q1\ s1.\ e2\ s1\ (\lambda q2\ s2.$
      $c\ (\lambda t.\ \text{union5}\ (q1\ s1)\ (q2\ t))\ s2))$

Again, the rule for parallel composition is more complicated. The grid structure is build in several steps:

1. The locations of both sequential graphs are extracted, filtering out the

initial and final location (sets $mid1$, $mid2$).

2. The inner portion of the grid is computed (sets $cross1$, $cross2$)

3. The side of each grid is computed (sets $side1$, $side2$, $side3$, $side4$), whilst carefully realising the renaming of $\langle s, s\rangle$ to s and $\langle t, t\rangle$ to t, in order to connect the graph to the remaining edges

Note that in the formalisation, the function $\langle\ ,\rangle$ is named $Choice$. Putting the computation of all these different sets together, the rule for parallel composition looks like this:

$$
\begin{array}{l}
par_{Prog->Prog->Prog} = \\
\quad \lambda e1\ e2.\ \lambda s\ c. \\
\quad\quad e1\ s\ (\lambda p1\ s1.\ e2\ s\ (\lambda p2\ s2. \\
\quad\quad\quad c\ (\lambda t.\ \textbf{let} \\
\quad\quad\quad\quad q1 = p1\ t \\
\quad\quad\quad\quad q2 = p2\ t \\
\quad\quad\quad\quad mid1 = \text{map4}\ (\lambda s1\ a\ b\ s2.\ s1)\ (\text{filter4}\ (\lambda s11\ a\ b\ s2.\ s11 \neq s)\ q1) \\
\quad\quad\quad\quad mid2 = \text{map4}\ (\lambda s1\ a\ b\ s2.\ s1)\ (\text{filter4}\ (\lambda s11\ a\ b\ s2.\ s11 \neq s)\ q2) \\
\quad\quad\quad\quad cross1 = \text{flatten1\_4}\ (map\ (\lambda s.\ \text{map4\_4}\ (\lambda s1\ a\ b\ s2.\ (\lambda q. \\
\quad\quad\quad\quad\quad q\ (Choice\ s\ s1)\ a\ b\ (Choice\ s\ s2)))\ q2)\ mid1) \\
\quad\quad\quad\quad cross2 = \text{flatten1\_4}\ (map\ (\lambda s.\ \text{map4\_4}\ (\lambda s1\ a\ b\ s2.\ (\lambda q. \\
\quad\quad\quad\quad\quad q\ (Choice\ s1\ s)\ a\ b\ (Choice\ s2\ s)))\ q1)\ mid2) \\
\quad\quad\quad\quad side1 = \text{mapCase4}\ (\lambda s11\ a\ b\ s2.\ s11 = s) \\
\quad\quad\quad\quad\quad (\lambda s11\ a\ b\ s22\ q.\ q\ s11\ a\ b\ (Choice\ s22\ s11)) \\
\quad\quad\quad\quad\quad (\lambda s11\ a\ b\ s22\ q.\ q\ (Choice\ s11\ s)\ a\ b(Choice\ s22\ s))\ q1 \\
\quad\quad\quad\quad side2 = \text{mapCase4}\ (\lambda s11\ a\ b\ s2.\ s11 = s) \\
\quad\quad\quad\quad\quad (\lambda s11\ a\ b\ s22\ q.\ q\ s11\ a\ b\ (Choice\ s11\ s22)) \\
\quad\quad\quad\quad\quad (\lambda s11\ a\ b\ s22\ q.\ q\ (Choice\ s\ s11)\ a\ b\ (Choice\ s\ s22))\ q2 \\
\quad\quad\quad\quad side3 = \text{mapCase4}\ (\lambda s1\ a\ b\ s22.\ s22 = t) \\
\quad\quad\quad\quad\quad (\lambda s11\ a\ b\ s22\ q.\ q\ (Choice\ s11\ s22)\ a\ b\ s22) \\
\quad\quad\quad\quad\quad (\lambda s11\ a\ b\ s22\ q.\ q\ (Choice\ s11\ t)\ a\ b\ (Choice\ s22\ t))\ q1 \\
\quad\quad\quad\quad side4 = \text{mapCase4}\ (\lambda s1\ a\ b\ s22.\ s22 = t) \\
\quad\quad\quad\quad\quad (\lambda s11\ a\ b\ s22\ q.\ q\ (Choice\ s22\ s11)\ a\ b\ s22) \\
\quad\quad\quad\quad\quad (\lambda s11\ a\ b\ s22\ q.\ q\ (Choice\ t\ s11)\ a\ b\ (Choice\ t\ s22))\ q2 \\
\quad \textbf{in}\ \text{union5}\ cross1(\text{union5}\ cross2(\text{union5}\ side1 \\
\quad\quad (\text{union5}\ side2(\text{union5}\ side3side4))))\ (Par\ s)))
\end{array}
$$

4.2.2 Transition System Unfolding And Satisfiability

The algorithm for unfolding a program graph into a transition system was already presented in section 2. Due to its use of unbounded recursion, translating the unfolding function into a lambda term requires some extra thought: We argued that the algorithm must terminate because the set of memory configurations is finite and therefore so is the set of states. This termination argument is not intrinsic to the algorithm itself, but we can make it intrinsic by explicitly limiting the number of iterations it goes through.

In each iteration, before the algorithm may terminate, at least one previously unknown transition/state must be discovered. As an upper bound, we can therefore use the cardinality of the state space domain $|Loc \times Mem|$. The idea now is to supply a function $UNFOLD$ for exactly one iteration and iterate it n times, such that the complete unfolding function is given by $UNFOLD^n$. In this scenario, n is our upper bound and must be computed by HOLMCF for a given input program and an appropriate iterator function must be supplied. We show an unfolding function for the case $n = 5$:

type $TS = Set4\ Loc\ Mem\ Loc\ Mem$

iterator : $(\alpha \to \alpha) \to (\alpha \to \alpha) =$
  $\lambda f.\ \lambda x.\ f\ (f\ (f\ (f\ (f\ x))))$

unfold1 : $PG \to (TS \to TS \to TS) \to TS \to TS \to TS =$
  $\lambda pg.\ \lambda r.\ \lambda old\ active.$
    let new_states = flatten1_4 (map4 ($\lambda q0\ z0\ q\ z.$
        let active_trans = filter4 ($\lambda s1\ a\ uf\ s2.\ s1 = q \wedge az$) pg
        in map4_4 ($\lambda s1\ a\ uf\ s2.(\lambda q.\ q\ s1\ z\ s2\ (uf\ z))$) active_trans) $active$)
      $new2$ = filter4 ($\lambda l1\ s1\ l2\ s2.\ \neg$(member4 $l1\ s1\ l2\ s2\ old$)) new_states
    in r (union4 $new2\ old$) $new2$

unfold : $Prog \to Mem \to TS =$
  $\lambda pg\ s0.$ (iterator (unfold1 ($pg\ Start\ (\lambda c\ s.\ c\ End)$)) ($\lambda b\ c.\ b$))
    empty4 (singleton4 $Start\ s0\ Start\ s0$)

Here the function *iterator* must be generated by HOLMCF. A regrettable sacrifice which was made to translate the unfolding algorithm to the term-level is the omission of the if-then-else clause to test whether new_trans is empty, and if so, to stop the iteration early. If-then-else clauses can be axiomatised, but the current provers do not apply axioms during the normalisation stage, which will result in massive terms of untreated if-then-else clauses. There has however been research to implement support for if-then-else in HOL provers. [3]

Stating the iterator function naively like above is generally impractical and some better scheme is required to make it space-efficient (that is logarithmic instead of linear in the size of the term). Devising such a scheme remains future work.

We now put in the final piece of the puzzle by axiomatising the set of paths over a transition system and the $\models$-relation for a program and an LTL formula:

$$
\begin{array}{l}
is\_path : TS \to Mem \to (Nat \to Loc) \to (Nat \to Mem) \to o = \\
\quad \lambda TS.\ \lambda m0.\ \lambda p1\ p2. \\
\qquad (p1\ \text{zero} = Start \wedge p2\ \text{zero} = m0) \\
\qquad \wedge (\forall i.\ \text{member4}\ (p1\ i)\ (p2\ i)\ (p1\ (\text{succ}\ i))\ (p2\ (\text{succ}\ i))\ TS)
\end{array}
$$

$$
\begin{array}{l}
models : PG \to Mem \to Form \to o = \\
\quad \lambda PG\ m0\ f.\ \forall p1\ p2.\ \text{is\_path}\ PG\ m0\ p1\ p2 \Rightarrow f\ p2
\end{array}
$$

Note that we cannot quantify over paths $Nat \to Loc \times Mem$ directly, due to the lack of pair types in simply typed lambda calculus. When modeling sets of tuples we resolved this issue through currying. This time, we can make use of the following law in constructive logic to work around it:

$$A \to (B \times C) \simeq (A \to B) \times (A \to C)$$

The two functions we quantify over instead are called $p1$ and $p2$.

5 Extended Normalisation

5.1 Automation Issues

We have discusssed in detail all the necessary devices to formulate model checking problems in HOL. It was already hinted towards the fact that, as of now, automated proof systems have great difficulty with the presented approach. This section deals with the most evident problem of term normalisation.

The formulation presented in this thesis encodes imperative programs as λ-terms which reduce to transition systems through β-normalisation. However, the representation of sets as an extension of Curch's characteristic functions does not only make use of λ-abstraction and application but also of logical formulae. The lack of treating these formulae during normalisation is responsible for a problematic inflation of the term sizes through leaving parts of the terms unsimplified. Consider the following example:

$$\begin{aligned}
&\emptyset \cup \{a, b\} \cup \{a, c\} \\
&= \text{union } (\lambda f.\ \bot)\ (\text{union } (\lambda f.\ f\ a \vee f\ b)\ (\lambda f.\ f\ a \vee f\ c)) \\
&\rightarrow_\beta (\lambda f.\ \bot \vee a \vee b \vee a \vee c)
\end{aligned}$$

What we really wish for is a reduction to the term $(\lambda f.\ a \vee b \vee c)$ without the duplicate a and irrelevant constant $\bot$. Notice that without the necessary simplifications, the union of two identical sets doubles the size of the resulting set!

A second problem is posed by the late treatment of equality:

$$\begin{aligned}
&\{l \mid l \in \{\text{Start}, \text{Seq Start}, \text{Seq (Seq Start)}\} \wedge l = \text{Seq Start}\} \\
&= \text{filter } (\lambda l.\ l = \text{Seq Start})\ (\lambda f.\ f\ \text{Start} \vee f\ \text{Seq Start} \vee f\ \text{Seq (Seq Start)}) \\
&\rightarrow_\beta (\lambda f.\ (\text{Start} = \text{Seq Start} \wedge f\ \text{Start}) \\
&\qquad\qquad \vee (\text{Seq Start} = \text{Seq Start} \wedge f\ \text{Seq Start}) \\
&\qquad\qquad \vee (\text{Seq (Seq Start)} = \text{Seq Start} \wedge f\ \text{Seq (Seq Start)}))
\end{aligned}$$

In this scenario the filter function made the term grow instead of letting it shrink. The final term one expects should rather be $(\lambda f.f\ \text{Seq Start})$. But the prove systems cannot conclude the intended form purely by normalisation, they would need to apply the supplied distinctiveness lemmata as well. What is therefore required would be a setting to make different constant symbols distinct by default or to recognise such lemmata beforehand and use this obtained knowledge in the normalisation phase.

5.2 Normalisation of Equations & Propositional Formulae

In this thesis, we chose the former (much simpler) approach of making different constant symbols distinct and modify the normalisation routine in *Satallax* to behave accordingly. Satallax is a good candidate for such a modification, because its code base is rather small. The normalisation code was then ported later to HOLMCF to use other provers with the normalised problems. This is also the reason why we present the routine based on Satallax' logical signature of $\{\bot, \Rightarrow\}$ instead of the more common choie of $\{\neg, \wedge, \vee\}$.

For β-normalisation, we use the standard procedure of *normal order reduction* to normal form. A λ-term is in normal form if and only if it contains no redexes, which are subterms of shape $((\lambda x.e_1)\ e_2)$.

It is important to note that Satallax uses de-Brujin-indices [13] for representing λ-terms and we therefore adopt this notion. When using de-Brujin-indices, variable names following λ-binders are omitted, and variables appearing in the terms themselves are replaced by numerals, where a numeral n says that the variable it represents is bound to the n-th outermost λ-symbol. For instance, λ-terms $(\lambda x.\lambda y.x)$ and $(\lambda x.\lambda y.y)$ are written as $(\lambda(\lambda 1))$ and $(\lambda(\lambda 0))$ respectively. While substitution in the context of de-Brujin-indices is a bit more involved, tests for α-equivalence (term equality modulo variable renaming) become very efficient.

A simple function to realise a reduction to normal form is the following one, taken from [25]:

```
cbn m = case m of
    (La x e) → m
    (Ap e1 e2) →
        case cbn e1 of
            (La x e) → cbn (subst e2 e1)
            e3 → Ap e3 e2
    _ → m

norm m = case m of
    (La x e) → La x (norm e)
    (Ap e1 e2) →
        case cbn e1 of
            (La x e) → norm (subst e2 e1)
            e3 → Ap (norm e3) (norm e2)
    _ → m
```

The procedure works by exhaustively reducing the outermost redexes (function *cbn*) and, if none are left, to continue this process in the subterms (function *norm*). A subtle but vital optimisation is to normalise substitutes before insertion, i.e. the substitution function (*subst e*2) should be replaced

by (*subst* (*norm* *e*2)) in both cases. The resulting evaluation strategy is *strict* evaluation, where arguments are always evaluated before use (if they are used at all). This strategy is still complete due to the guarantee of termination in the presence of types. It might further be refined to lazy evaluation with memoisation to possibly gain additional performance.

We now proceed to extend the presented function *norm* to handle logical formulae as well. We first address equations. Naturally, the most significant amount of equations can only be solved by the inference engine of the prover, but we can use sufficient approximations for solving those which occur in a functional programming context. As a first step, formulae A and B in a term $A = B$ should be tested for α-equivalence. If they are, $A = B$ reduces to $\top$. Secondly, if two distinct constant symbols are compared, we reduce to $\top$ if they are both the same and $\bot$ otherwise. If two distinct free variables are detected, for instance in the expression $a\ X = a\ Y$, we can not reduce the equation further and leave it unmodified. The implemented function *normEq* thus has return type *MaybeBool* containing values $\{\text{Just}\ \top, \text{Just}\ \bot, \text{Nothing}\}$, where Nothing is returned if no statement about the terms can be made due to the presence of free variables.

Applications are handled in the following way: 1.) Either their head symbols (the left most symbol in a chain "(((... a) b) c)") are unequal or their equality is unknown, at which point the comparison can be stopped, or 2.) they are equal and their arguments must be compared. In the latter case, both terms must have been applied to the same number of arguments to preserve well-typedness. An appropriate representation of terms to do this comparison of the two argument lists is the *spine notation* presented in [14]. Essentially, instead of nesting applications leftwards (as in "(((... a) b) c)"), applications are nested rightwards (as in "(a (b (c...)))") as a typical singly-linked list, thus making the head symbol easily accessible. We require this easy access, because the head symbols of two terms must be compared first to determine if we need to look at their arguments. In our case, because the head symbol is treated differently, it is stored separately from its arguments. A term $(((f\ a)\ b)\ c)$ is thus stored as a pair $(f, [a, b, c])$

```
toSpine (Ap a b) args = toSpine a (args ++ [b])
toSpine x args = (x, args)

-- compare argument lists
argEq [] [] 0 = Just True
argEq [] [] 1 = Nothing
argEq (a:as) (b:bs) i =
    case trmEq2 (toSpine a []) (toSpine b []) of
        Just True → argEq as bs i
        Just False → Just False
        Nothing → argEq as bs 1
```

```
-- compare head symbols
trmEq1 n1 n2 =
    case (n1, n2) of
        (F, F) → Just True
        (I, I) → Just True
        (La x, La y) → trmEq2 (toSpine x []) (toSpine y [])
        (DB i, DB j) → if i == j then Just True else Nothing
        (Name x, Name y) → Just (x == y)
        (x, y) → Nothing

-- compare spines
trmEq2 (F, []) (I, [F, F]) = Just False
trmEq2 (I, [F, F]) (F, []) = Just False
trmEq2 (a, as) (b, bs) =
    case trmEq1 a b of
        Just True → argEq as bs 0
        Just False → Just False
        Nothing → Nothing
```

5.3 Summary & Discussion

The most important part of the extended normalisation routine concerns the simplification of propositional formulae. To this end, duplicates are removed from any detected sequence of conjunctions $A_1 \wedge A_2 \wedge ... \wedge A_n$ or sequence of disjunctions $B_1 \vee B_2 \vee ... \vee B_n$, and occurrences of the neutral elements of these connectives, $\top$[5] and $\bot$, are filtered out. Likewise, occurrences of absorbing elements, $\top$ for $\wedge$ and $\bot$ for $\vee$ reduce whole chains to just $\top$ or $\bot$. In Satallax' signature $\{\bot, \Rightarrow\}$, conjunctions map to negated implications and disjunctions map to bare implications, like this:

$$A \wedge B \equiv (A \Rightarrow \overline{B}) \Rightarrow \bot \quad A \vee B \equiv \overline{A} \Rightarrow B$$

$$\overline{A \wedge B} \equiv (A \Rightarrow \overline{B}) \qquad \overline{A \vee B} \equiv (\overline{A} \Rightarrow B) \Rightarrow \bot$$

Therefore, whenever we see a negated implication, we know we are dealing with a sequence conjunctions (of minimal length 2, if no further nested conjunctions are found), in the case of an implication we are dealing with a disjunction. Because formulae $(A \Rightarrow B) \Rightarrow \bot$ and $A \Rightarrow B$ in fact map to $A \wedge \neg B$ and $\neg A \vee B$, the subformulae preceded by a $\neg$ are overlined to mark that they are of opposite polarity or, in other words, that they are *inverted*. For inverted formulae, the rules are switched around: If we are looking at an inverted negated implication, we are dealing with a disjunction, in the case of an inverted implication, we are dealing with a conjunction. We adopt Smullyan's [26] terminology and call conjunctive formulae α-formulae

[5] $\top$ is defined as $\bot \Rightarrow \bot$ in Satallax

and disjunctive formulae β-formulae. The simplification functions are called simp-α and simp-β respectively. Note that the notion of a β-formula is completely unrelated to the idea of β-reduction in lambda calculus!

```
simp−α m pol = case m of
    ((e1 ⇒ e2) ⇒ ⊥) →
        if pol = 1
            then { norm (e1 ⇒ e2) }
            else simp−α e1 pol ∪ simp−α e2 (1−pol)
    (e1 ⇒ e2) →
        if pol = 1
            then simp−α (neg m) 0
            else { norm m }
    c → case cbn c of
            c@(_ ⇒ _) → simp−α c pol
            c → if pol = 1
                    then case norm c of
                            (x ⇒ F) → { x }
                            c → { neg c }
                    else { norm c }

simp−β m pol = case m of
    ((e1 ⇒ e2) ⇒ ⊥) →
        if pol = 1
            then simp−β (e1 ⇒e2) 0
            else { norm m }
    (e1 ⇒ e2) →
        if pol = 1
            then { norm (neg m) }
            else simp−β e1 (1−pol) ∪ simp−β e2 pol
    c → case cbn c of
            c@(_ ⇒ _) → simp−β c pol
            c → if pol = 1
                    then case norm c of
                            (x ⇒ ⊥) → { x }
                            c → { neg c }
                    else { norm c }
```

Finally, the preprocessing of equations and propositional formulae can be combined with the usual β-normalisation. The normalisation procedure corresponds to the same *normal order reduction* function presented earlier with the minor (but vital) refinement of normalising substitutes before substitution and additional cases to handle α-formulae, β-formulae and equations.

```
fold−α a@(a2 ⇒⊥) b@(b2 ⇒ ⊥) = (a ⇒ b2) ⇒ ⊥
fold−α a          b@(b2 ⇒ ⊥) = (a ⇒ b2) ⇒ ⊥
fold−α a@(a2 ⇒⊥) b           = (a ⇒ (b ⇒ ⊥)) ⇒ ⊥
fold−α a          b           = (a ⇒ (b ⇒ ⊥)) ⇒ ⊥

fold−β a@(a2 ⇒⊥) b@(b2 ⇒ ⊥) = a2 ⇒ b
```

```
fold−β a          b@(b2 ⇒ ⊥) = (a ⇒ ⊥) ⇒ b
fold−β a@(a2 ⇒⊥) b           = a2 ⇒ b
fold−β a          b           = (a ⇒ ⊥) ⇒ b

norm m = case m of
    ((e1 ⇒ e2) ⇒ ⊥) →
        let r = filter (≠ (⊥ ⇒ ⊥)) (simp−α m 0)
        in if r = ∅
            then ⊥ ⇒ ⊥
            else if ⊥ ∈ r
                then ⊥
                else fold1 fold−α r
    (e1 ⇒ e2) →
        let r = filter (≠ ⊥) (simp−β m 0)
        in if r = ∅
           then ⊥
           else if (⊥ ⇒ ⊥) ∈ r
                    then ⊥ ⇒ ⊥
                    else fold1 fold−β r
    (e1 = e2) →
        let n1 = norm e1
            n2 = norm e2
        in case trmEq2 (toSpine n1 []) (toSpine n2 []) of
            Just True → true
            Just False → F
            Nothing → (n1 = n2)
    (La e) → La (norm e)
    (Ap e1 e2) →
        case cbn e1 of
            (La e) → norm (subst e (norm e2))
            e3 → let e4 = norm e3
                 in Ap e4 (norm e2)
    _ → m
```

The presented extension for the normalisation of HOL formulae reduces the program terms to transition systems as expected. It may be of use to solve problems in other domains as well, if they likewise require significant amounts of term level computation (for instance dealing with large Church sets of individual objects etc.). It is also more robust, in the sense that it deals well with a class of possibly trivial problems, which explode in size through naive β-normalisation.

On the other hand, the question may be raised whether Church sets and other means of data representation, which exploit the availability of logical connectives, are at all an appropriate technique for term-level computation. One can argue that Boolean connectives themselves can be encoded in lambda calculus. If one would replace all the in-built logical connectives used for Church sets with the well-known encodings of Boolean operations in

lambda calculus, the problem addressed here would not turn up in the first place! However, this again can not be done due to the restrictive nature of the type system, because these encodings require higher-rank polymorphic types.

6 Conclusion

6.1 Results

HOLMCF was tested on a range of very small example programs with not more than maybe a few dozen states. It is noteworthy that these problems could be solved almost instantaneously by first-order systems E and Vampire, while higher-order systems Satallax and LEO-II took up to a few minutes to solve the same set of problems or could not solve them at all within reasonable time.

As a more interesting proof-of-concept, Vampire managed to proof the mutual exclusion property of *Peterson's Algorithm* [21] as a HOLMCF-generated problem within a few seconds. Including the computation time of HOLMCF (most notably the normalisation process) and the time used to translate the problem to first-order logic with Nitpick, the total time used still remains well under a minute. All experiments were carried out using SystemOnTPTP[6].

This underscores that to this day the development of higher-order prove systems is still at an early stage and that the potential for the improvement of these systems is substantial. Ideally, they should outperform or at least keep up with the first-order systems.

At the same time, it shows the feasibility of the embedding technique for very small real-word problems. If these problems are small enough and the verification logic turns out to be very domain-specific, it might be sufficient to devise respective embeddings instead of developing more complex (most likely automata-based) decision procedures. Another advantage to this approach is that if the semantics of the model language are already formalised through the state-space generation procedure, it is a natural further step to prove its correctness.

6.2 Related & Future Work

The idea of using embeddings to automate verification logics and the model checking process using automated theorem provers is a novel one, but there has been research on the formalisation of model checking in HOL in general. Esparza et al. [15] present a traditional automata-based LTL model checker, which has been proven correct in Isabelle/HOL. Schneider and Hoffmann [23] present a conversion scheme for translating LTL formulae to ω-automata,

[6] SystemOnTPTP is an online service which offers remote access to a range of different theorem provers and related tools, http://www.cs.miami.edu/~tptp/cgi-bin/SystemOnTPTP

which are formalised in HOL. The motivation behind this work is the incorporation of symbolic model checkers into HOL as a tactic.

For the future, it would be desirable to have more verification logics to chose from, most notably an embedding of Computation Tree Logic (CTL), which might be just as popular as LTL. In addition, it would help to have more syntactical features to improve the look of embedded formulae.

Above all else, this thesis motivates the advancement of higher-order theorem provers, both in terms of strength as well as expressivity. Simple Type Theory's term language is too weak a formalism to model objects as they appear in everyday mathematics. From a user's perspective, System F, which is a lambda calculus with arbitrary-rank type polymorphism, seems like a highly desirable choice. Alternatively, sum and product types, as they are featured in most interactive HOL proof assistants, would already result in a substantial improvement in expressivity. Although the formalisations presented in this thesis work as intended, they occasionally lack clarity and elegance in some parts in order to cope with the weaknesses of STT. Of all the compromises, the outsourcing of the normalisation process is probably the one most conflicting with the original intents of this thesis, but was vital to make it work. On the other hand however, HOLMCF may be used in the future for creating interesting benchmark problems and thus aid in the improvement of these tools.

A Modeling language syntax

```
Model : Decls2 StmntPar Props

Props : property '{' Form '}' Props
      | {- empty -}

StmntPar : StmntSeq '|' StmntPar
         | StmntSeq
         | { Skip }

StmntSeq : Stmnt StmntSeq
         | Stmnt

Stmnt : while '(' Disj ')' Stmnt
      | while '(' ')' Stmnt
        -- dangling else fix with invisible terminal symbol
      | if '(' Disj ')' Stmnt %prec "then"
      | if '(' Disj ')' Stmnt else Stmnt
      | for '(' Assgnmnt ';' Disj ';' Assgnmnt ')' Stmnt
      | run '(' Assgnmnt ';' Disj ';' Assgnmnt ')' Stmnt
      | Assgnmnt ';'
      | ';'
      | Block

Decl : var '=' val
     | var

DeclLst : Decl ',' DeclLst
        | Decl ';'

Decls2 : int DeclLst Decls2
       | {- empty -}

Block : '{' Decls2 StmntPar '}'

Assgnmnt : var '=' Exp

Disj  : Disj '||' Conj
      | Conj

Conj  : Conj '&&' Lit
      | Lit

Lit       : '!' Lit
      | var
      | Exp '<' Exp
      | Exp '>' Exp
      | Exp '==' Exp

Exp   : Exp '+' Term
      | Exp '-' Term
      | Term
```

```
Term  : Term '*' Factor
      | Term '/' Factor
      | Factor

Factor : val
       | var
       | '(' Exp ')'
```

B Embedding language syntax

```
Emb : Decls

Decls : basetype var ';' Decls
      | type var TypePars'=' FunType ';' Decls
      | axiom Form ';' Decls
      | var ':' FunType '=' Form ';' Decls
      | var ':' FunType ';' Decls
      | export '{' Decls '}' Decls
      | {- empty -}

TypePars : var TypePars
         | {- empty -}

FunType : Type '->' FunType
        | Type

Type : var TypeArgs
     | '(' FunType ')'

TypeArgs : var TypeArgs
         | '(' FunType ')' TypeArgs
         | {- empty -}

LetBindings : var '=' Form ';' LetBindings
            | var '=' Form ';'

Form : '\\' Args '.' Form
     | '!' Args '.' Form
     | '?' Args '.' Form
     | let LetBindings in Form
     | FDisj

FDisj : FDisj '|' FConj
      |  FDisj '=>' FConj
      | FConj

FConj : FConj '&' FCmp
      | FCmp

FCmp : FCmp '==' App
```

```
      | FCmp '~=' App
      | App

App : App Simp
    | Simp

Simp : var
     | '~' Simp
     | '(' Form ')'
     | true
     | false

Args : var Args
     | '*' Args
     | var
     | '*'
```

References

[1] Peter B. Andrews. General models, descriptions, and choice in type theory. *J. Symb. Log.*, 37(2):385–394, 1972.

[2] Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. Tps: A theorem-proving system for classical type theory. *Journal of Automated Reasoning*, 16(3):321–353, 1996.

[3] Julian Backes and Chad E. Brown. Analytic tableaux for higher-order logic with choice. *J. Autom. Reasoning*, 47(4):451–479, 2011.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, Cambridge, MA, USA, 2008.

[5] C. Benzmüller, L. Paulson, F. Theiss, and A. Fietzke. Progress report on LEO-II – an automatic theorem prover for higher-order logic. In *TPHOLs 2007 Emerging Trends Proceedings*, pages 33–48. Internal Report 364/07, Department of Computer Science, University Kaiserslautern, Germany, 2007.

[6] Christoph Benzmüller. Automating quantified conditional logics in HOL. In Francesca Rossi, editor, *23rd International Joint Conference on Artificial Intelligence (IJCAI-13)*, pages 746–753, Beijing, China, 2013.

[7] Christoph Benzmüller, Maximilian Claus, and Nik Sultana. Systematic Verification of the Modal Logic Cube in Isabelle/HOL. In Cezary Kaliszyk and Andrei Paskevich, editors, *PxTP 2015*. EPTCS, 2015. To appear.

[8] Christoph Benzmüller, Florian Rabe, and Geoff Sutcliffe. THF0 – the core of the TPTP language for classical higher-order logic. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, IJCAR 2008*, volume 5195 of *LNCS*, pages 491–506. Springer, 2008.

[9] Christoph Benzmüller, Frank Theiss, Larry Paulson, and Arnaud Fietzke. LEO-II - a cooperative automatic theorem prover for higher-order logic. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *LNCS*, pages 162–170. Springer, 2008.

[10] Christoph Benzmüller and Bruno Woltzenlogel Paleo. Automating Gödel's ontological proof of God's existence with higher-order automated theorem provers. In Torsten Schaub, Gerhard Friedrich, and

Barry O'Sullivan, editors, *ECAI 2014*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 93 – 98. IOS Press, 2014.

[11] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A ounterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving (ITP 2010)*, volume 6172, pages 131–146, 2010.

[12] Chad E. Brown. Satallax: An automated higher-order prover. In Bernhard Gramlich, Dale Miller, and Ulrike Sattler, editors, *6th International Joint Conference on Automated Reasoning (IJCAR 2012)*, pages 111 – 117. Springer, 2012.

[13] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *INDAG. MATH*, 34:381–392, 1972.

[14] Iliano Cervesato and Frank Pfenning. A linear spine calculus. Technical report, Journal of Logic and Computation, 2003.

[15] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A fully verified executable ltl model checker. In N. Sharygina and H. Veith, editors, *Computer Aided Verification (CAV 2013)*, volume 8044, pages 463–478, 2013.

[16] Leon Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15(2):81–91, 1950.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[18] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[19] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV)*, pages 1–35, 2013.

[20] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[21] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[22] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[23] Klaus Schneider and Dirk W. Hoffmann. A HOL conversion for translating linear time temporal logic to omega-automata. In *Theorem Proving*

in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings, pages 255–272, 1999.

[24] Stephan Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2,3):111–126, August 2002.

[25] Peter Sestoft. Demonstrating lambda calculus reduction. In Torben ÆMogensen, David A. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation*, pages 420–435. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[26] Raymond M. Smullyan. *First-Order Logic*. Dover Publications, New York, 1995.