Chapter 5

Decidable Fragments

In previous chapters we have concentrated on the basic laws of reasoning and their relationship to types and programming languages. The logics and type theories we considered are very expressive. This is important in many applications, but it has the disadvantage that the question if a given proposition is true is undecidable in general. That is, there is no terminating mechanical procedure which, given a proposition, will tell us whether it is true or not. This is true for first-order logic, arithmetic, and more complex theories such as the theory of lists. Furthermore, the proof (which we do not have time to consider in this course), does not depend on whether we allow the law of excluded middle or not.

In programming language application, we can sometimes work around this limitation, because we are often not interested in theorem proving, but in type-checking. That is, we are given a program (which corresponds to a proof) and a type (which corresponds to a proposition) and we have to check its validity. This is a substantially easier problem than deciding the truth of a proposition—essentially we have to verify the correctness of the applications of inference rules, rather than to guess which rules might have been applied.

However, there are a number of important applications where we would like to solve the theorem proving problem: given a proposition, is it true. This can come about either directly (verify a logic property of a program or system) or indirectly (take a general problem and translate it into logic).

In this chapter we consider such applications of logic, mostly to problems in computer science. We limit ourselves to fragments of logic that can be mechanically decided, that is, there is a terminating algorithm which decides whether a given proposition is true or not. This restricts the set of problems we can solve, but it means that in practice we can often obtain answers quickly and reliably. It also means that in principle these developments can be carried out within type theory itself. We demonstrate this for our first application, based on quantified Boolean formulas.

The material here is not intended as an independent introduction, but complements the treatment by Huth and Ryan [HR00].

5.1 Quantified Boolean Formulas

In Section 3.6 we have seen the data type **bool** of Booleans with two elements, **true** and **false**. We have seen how to define such operations as boolean negation, conjunction, and disjunction using the elimination rule for this type which corresponds to an if-then-else construct. We briefly review these constructs and also the corresponding principle of proof by cases. In accordance with the notation in the literature on this particular subject (and the treatment in Huth and Ryan [HR00]), we write 0 for **false**, 1 for **true**, $b \cdot c$ for *and*, $b + c$ for *or*, and $\overline{b}$ for *not*.

The Boolean type, **bool**, is defined by two introduction rules.

$$\frac{}{\mathbf{bool}\ \mathit{type}}\ \mathbf{bool}F$$

$$\frac{}{\Gamma \vdash 0 \in \mathbf{bool}}\ \mathbf{bool}I_0 \qquad \frac{}{\Gamma \vdash 1 \in \mathbf{bool}}\ \mathbf{bool}I_1$$

The elimination rule follows distinguishes the two cases for a given Boolean value.

$$\frac{\Gamma \vdash b \in \mathbf{bool} \qquad \Gamma \vdash s_1 \in \tau \qquad \Gamma \vdash s_0 \in \tau}{\Gamma \vdash \mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_0 \in \tau}\ \mathbf{bool}E$$

The reduction rules just distinguish the two cases for the subject of the **if**-expression.

$$\begin{array}{rcl} \mathbf{if}\ 0\ \mathbf{then}\ s_1\ \mathbf{else}\ s_0 & \Longrightarrow & s_0 \\ \mathbf{if}\ 1\ \mathbf{then}\ s_1\ \mathbf{else}\ s_0 & \Longrightarrow & s_1 \end{array}$$

Now we can define typical functions on booleans, such as *and*, *or*, and *not*, transcribing their truth tables. We make no attempt here to optimize the definitions, but simply distinguish all possible cases for the inputs.

$$\begin{array}{rcl} x \cdot y & = & \mathbf{if}\ x\ \mathbf{then}\ (\mathbf{if}\ y\ \mathbf{then}\ 1\ \mathbf{else}\ 0)\ \mathbf{else}\ (\mathbf{if}\ y\ \mathbf{then}\ 0\ \mathbf{else}\ 0) \\ x + y & = & \mathbf{if}\ x\ \mathbf{then}\ (\mathbf{if}\ y\ \mathbf{then}\ 1\ \mathbf{else}\ 1)\ \mathbf{else}\ (\mathbf{if}\ y\ \mathbf{then}\ 1\ \mathbf{else}\ 0) \\ \overline{x} & = & \mathbf{if}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ 1 \end{array}$$

Following this line of thought, it is quite easy to define universal and existential quantification over Booleans. The idea is that $\forall x{\in}\mathbf{bool}.\ f(x)$ where f is a Boolean term dependent on x is representated by $\mathit{forall}(\lambda x{\in}\mathbf{bool}.\ f(x))$ so that

$$\begin{array}{rcl} \mathit{forall} & \in & (\mathbf{bool} \to \mathbf{bool}) \to \mathbf{bool} \\ \mathit{forall} & = & \lambda f{\in}\mathbf{bool} \to \mathbf{bool}.\ f\ 0 \cdot f\ 1 \end{array}$$

Somewhat more informally, we write

$$\forall x.\ f(x) \quad = \quad f(0) \cdot f(1)$$

but this should only be considered a shorthand for the above definition. The existential quantifier works out similarly, replacing conjunction by disjunction. We have

$$\begin{array}{rcl} \mathit{exists} & \in & (\mathbf{bool} \to \mathbf{bool}) \to \mathbf{bool} \\ \mathit{exists} & = & \lambda f{\in}\mathbf{bool} \to \mathbf{bool}.\ f\ 0 + f\ 1 \end{array}$$

or, in alternate notation,

$$\exists x.\, f(x) \quad = \quad f(0) + f(1)$$

The resulting language is that of *quantified Boolean formulas*. As the definitions above show, the value of each quantified Boolean formula (without free variables) can simply be computed, using the definitions of the operations in type theory.

Unfortunately, such an implementation is extremely inefficient, taking exponential time in the number of quantified variables, not only in the worst, but the typical case. Depending on the operational semantics we employ in type theory, the situation could be even worse, require exponential time in every case.

There are two ways out of this dilemma. One is to leave type theory and give an efficient imperative implementation using, for example, ordered binary decision diagrams as shown in Huth and Ryan. While this does not improve worst-case complexity, it is practical for a large class of examples.

Another possibility is to replace computation by reasoning. Rather than *computing* the value of an expression, we *prove* that it is equal to 1 or 0. For example, it is easy to prove that $(\forall x_1 \ldots \forall x_n.\ x_1 \cdots x_n \cdot 0) =_B 0$ even though it may take exponential time to compute the value of the left-hand side of the equation. In order to model such reasoning, we need the propositional counterpart of the elimination rule for **bool**.

$$\frac{\Gamma \vdash b \in \mathbf{bool} \qquad \Gamma \vdash M_1 : A(1) \qquad \Gamma \vdash M_0 : A(0)}{\Gamma \vdash \mathbf{case}\ b\ \mathbf{of}\ 1 \Rightarrow M_1 \mid 0 \Rightarrow M_0 : A(t)}\ \mathbf{bool}E$$

The rules for propositional equality between Booleans follow the pattern established by equality on natural numbers and lists.

$$\frac{\Gamma \vdash b \in \mathbf{nat} \qquad \Gamma \vdash c \in \mathbf{nat}}{\Gamma \vdash b =_B c\ prop} =_B F$$

$$\frac{}{\Gamma \vdash 0 =_B 0\ true} =_B I_0 \qquad \frac{}{\Gamma \vdash 1 =_B 1\ true} =_B I_1$$

$$no =_B E_{00}\ \textit{elimination rule} \qquad no =_B E_{11}\ \textit{elimination rule}$$

$$\frac{\Gamma \vdash 0 =_B 1\ true}{\Gamma \vdash C\ true} =_B E_{01} \qquad \frac{\Gamma \vdash 1 =_B 0\ true}{\Gamma \vdash C\ true} =_B E_{10}$$

As a simple example, we prove that for every $b \in \mathbf{bool}$ we have $b \cdot 0 =_B 0$. The proof proceeds by cases on b.

Case: $b = 0$. Then we compute $0 \cdot 0 =_B 0$ so by conversion we have $0 \cdot 0 =_N 0$.

Case: $b = 1$. Then we compute $1 \cdot 0 =_B 0$ so by conversion we have $1 \cdot 0 =_N 1$

Note that we can use this theorem for arbitrarily complex terms b. Its transcription into a formal proof using the rules above is straightforward and omitted.

An interesting proposal regarding the combination of efficient computation (using OBDDs) and proof has been made by Harrison [Har95]. From a trace of the operation of the OBDD implementation we can feasibly extract a proof in terms of the primitive inference rules and some other derived rules. This means what we can have a complex, optimizing implementation without giving up the safety of proof by generating a proof object rather than just a yes-or-no answer.

5.2 Boolean Satisfiability

A particularly important problem is Boolean satisfiability (SAT):

> Given a Boolean formula $\exists x_1 \ldots \exists x_n.\ f(x_1, \ldots, x_n)$ without free variables where $f(x_1, \ldots, x_n)$ does not contain quantifiers. Is the formula equal to 1?

Alternatively, we can express this as follows:

> Given a quantifier-free Boolean formula $f(x_1, \ldots, x_n)$, is there an assignment of 0 and 1 to each of the variables x_i which makes f equal to 1?

SAT is an example of an NP-complete problem: it can be solved in nondeterministic polynomial time (by guessing and then checking the satisfying assignment), and every other problem in NP can be translated to SAT in polynomial time. What is perhaps surprising is that this can be practical in many cases. The dual problem of validity (deciding if $\forall x_1 \ldots \forall x_n.\ f(x_1, \ldots, x_n)$ is equal to 1) is also often of interest and is co-NP complete.

There are many graph-based and related problems which can be translated into SAT. As a simple example we consider the question of deciding whether the nodes of a graph can be colored with k colors such that no two nodes that are connected by an edge have the same color.

Assume we are given a graph with nodes numbered $1, \ldots, n$ and colors $1, \ldots, k$. We introduce Boolean variables c_{ij} with the intent that

$$c_{ij} = 1 \quad \text{iff} \quad \text{node } i \text{ has color } j$$

We now have to express that constraints on the colorings by Boolean formulas. First, each node as exactly one color. For each node $1 \leq i \leq n$ we obtain a formula

$$\begin{aligned} u_i \quad = \quad & c_{i1} \cdot \overline{c_{i2}} \cdots \overline{c_{ik}} \\ & + \overline{c_{i1}} \cdot c_{i2} \cdots \overline{c_{ik}} \\ & + \cdots \\ & + \overline{c_{i1}} \cdot \overline{c_{i2}} \cdots c_{ik} \end{aligned}$$

There are n such formulas, each of size $O(n \times k)$. Secondly we want to express that two nodes connected by an edge do not have the same color. For any two

nodes i and m we have

$$\begin{array}{lcll} w_{im} & = & \overline{c_{i1} \cdot c_{m1}} \cdots \overline{c_{ik} \cdot c_{mk}} & \text{if there is an edge between } i \text{ and } m \\ w_{im} & = & 1 & \text{otherwise} \end{array}$$

There are $n \times n$ such formulas, each of size $O(k)$ or $O(1)$. The graph can be colored with k colors if each of the u_i and w_{im} are satisfied simultaneously. Thus the satisfiability problem associated with a graph is

$$(u_1 \cdots u_n) \cdot (w_{11} \cdots w_{1n}) \cdots (w_{n1} \cdots w_{nn})$$

The total size of the resulting formula is $O(n^2 \times k)$ and contains $n \times k$ Boolean variables. Thus the translation is clearly polynomial.

5.3 Constructive Temporal Logic

An important application of logic is model-checking, as explained in Huth and Ryan. Another excellent source on this topic is the book by Clarke, Grumberg and Peled [CGP99].

Here we give a constructive development of a small fragment of CTL. I am not aware of any satisfactory constructive account for all of CTL. For linear time temporal logic, Davies [Dav96] gives an extension of the Curry-Howard isomorphism with an interesting application to *partial evaluation*.

In order to model temporal logic we need to relativize our main judgment A *true* to particular states. We have the following judgments:

s *state*	s is a state
$s \to s'$	we can transition from state s to s' in one step
$A \mathbin{@} s$	proposition A is true in state s

We presuppose that s and s' are states when we write $s \to s'$ and that A is a proposition and s a state when writing $A \mathbin{@} s$.

Now all logical rules are viewed as rules for reasoning entirely within a given state. For example:

$$\frac{A \mathbin{@} s \qquad B \mathbin{@} s}{A \wedge B \mathbin{@} s} \wedge I$$

$$\frac{A \wedge B \mathbin{@} s}{A \mathbin{@} s} \wedge E_L \qquad\qquad \frac{A \wedge B \mathbin{@} s}{B \mathbin{@} s} \wedge E_R$$

For disjunction and falsehood elimination, there are two choices, depending on whether we admit conclusions in an arbitrary state s' or only in s, the state in which we have derived the disjunction or falsehood. It would seem that both choices are consistent and lead to slightly different logics.

Next, we model the AX and EX connectives. AX A is true in state s if A is true in every successor state s'. To express "in every successor state" we introduce the assumption $s \to S'$ for a new *parameter* S'.

$$\dfrac{\dfrac{}{s \to S'}\,u \quad \vdots \quad A \mathbin{@} S'}{\mathrm{AX}\ A \mathbin{@} s}\,\mathrm{AX}I^{S',u}$$

The elimination rule allows us to infer that A is true in state s' if AX A is true in s, and s' is a successor state to s.

$$\dfrac{\mathrm{AX}\ A \mathbin{@} s \qquad s \to s'}{A \mathbin{@} s'}\,\mathrm{AX}E$$

It is easy to see that this elimination rule is locally sound.

The rules for the EX connective follow from similar considerations.

$$\dfrac{s \to s' \quad A \mathbin{@} s'}{\mathrm{EX}\ A \mathbin{@} s}\,\mathrm{EX}I$$

$$\dfrac{\mathrm{EX}\ A \mathbin{@} s \qquad \dfrac{}{s \to S'}\,u \quad \dfrac{}{A \mathbin{@} S'}\,w \quad \vdots \quad C \mathbin{@} r}{C \mathbin{@} r}\,\mathrm{EX}E^{S',u,w}$$

We can now prove general laws, such as

$$\mathrm{AX}\ (A \wedge B) \equiv (\mathrm{AX}\ A) \wedge (\mathrm{AX}\ B)$$

Such proofs are carried out parametrically in the sense that we do not assume any particular set of states or particular transition relations. Laws derived in this manner will be true for any particular set of states and transitions.

If we want to reason about a particular system (which is done in model-checking), we have to specify the atomic propositions, states, and transitions. For example, the system on page 157 in Huth and Ryan is represented by the assumptions

$$\begin{array}{l} p\ \mathit{prop}, q\ \mathit{prop}, r\ \mathit{prop}, \\ s_0\ \mathit{state}, s_1\ \mathit{state}, s_2\ \mathit{state}, \\ s_0 \to s_1, s_0 \to s_2, \\ s_1 \to s_0, s_1 \to s_2, \\ s_2 \to s_2 \end{array}$$

Unfortunately, this is not yet enough. We can think of the transition rules above as introduction rules for the $s \to s'$, but we also need elimination rules. Because of the nature of the AX and EX connectives, it appears sufficient if we can distinguish cases on the target of a transition.

$$\frac{s_0 \to s \quad A \mathbin{@} s_1 \quad A \mathbin{@} s_2}{A \mathbin{@} s}\, s_0 \to E$$

$$\frac{s_1 \to s \quad A \mathbin{@} s_0 \quad A \mathbin{@} s_2}{A \mathbin{@} s}\, s_1 \to E$$

$$\frac{s_2 \to s \quad A \mathbin{@} s_2}{A \mathbin{@} s}\, s_2 \to E$$

In general, this would have to be augmented with additional rules, for example, letting is infer anything from an assumption that $s_2 \to s_1$ if there is in fact no such transition.[1] Now we can prove, for example, that AX $r \mathbin{@} s_0$ as follows

$$\dfrac{\dfrac{\overline{s_0 \to S}\, u \quad \overline{r \mathbin{@} s_1} \quad \overline{r \mathbin{@} s_2}}{r \mathbin{@} S}\, s_1 \to E}{\mathrm{AX}\, r \mathbin{@} s_0}\, \mathrm{AX}I^{S,u}$$

Despite the finiteness of the sets of states and the transition relations, this logic is different from the classical logic usually used, because we do not assume the law of excluded middle. Of course, this can be done which brings us back to the usual interpretation of the logic.

It is not clear how to carry this constructive development forward to encompass other connectives such as AG, AF, EG, etc. The difficulty here is that paths are infinite, yet we need to reason about global or eventual truth along such paths. In the classical development this is easily handled by introducing appropriate laws for negation and least and greatest fixpoint operations on monotone state transformations. We are currently investigating type-theoretic expressions of this kind of reasoning using inductive and co-inductive techniques.

[1] For the connectives given here, I do not believe that this is necessary.

Bibliography

[CGP99] E.M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, 1999.

[CR36] Alonzo Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, May 1936.

[Dav96] Rowan Davies. A temporal logic approach to binding-time analysis. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

[Har95] John Harrison. Binary decision diagrams as a HOL derived rule. *The Computer Journal*, 38:162–170, 1995.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. Hitherto unpublished note of 1969, rearranged, corrected, and annotated by Howard.

[HR00] Michael R.A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2000.

[ML80] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holland, 1980.

[ML96] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[Oka99] Chris Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, July 1999.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 249–257, Montreal, Canada, June 1998. ACM Press.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In A. Aiken, editor, *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, January 1999.