Automatic Deduction for Theories of Algebraic Data Types

by

Igor A. Chikanian

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

May 2011

Clark Barrett

Abstract

In this thesis we present formal logical systems, concerned with reasoning about algebraic data types.

The first formal system is based on the quantifier-free calculus (outermost universally quantified). This calculus is comprised of state change rules, and computations are performed by successive applications of these rules. Thereby, our calculus gives rise to an abstract decision procedure. This decision procedure determines if a given formula involving algebraic type members is valid. It is shown that this calculus is sound and complete. We also examine how this system performs practically and give experimental results. Our main contribution, as compared to previous work on this subject, is a new and more efficient decision procedure for checking satisfiability of the universal fragment within the theory of algebraic data types.

The second formal system, called Term Builder, is the deductive system based on higher order type theory, which subsumes second order and higher order logics. The main purpose of this calculus is to formulate and prove theorems about algebraic or other arbitrary user-defined types. Term Builder supports proof objects and is both, an interactive theorem prover, and verifier. We describe the built-in deductive capabilities of Term Builder and show its consistency. The logic represented by our prover is intuitionistic. Naturally, it is also incomplete and undecidable, but its expressive power is much higher than that of the first formal system. Among our achievements in building this theorem prover is an elegant and intuitive GUI for building proofs. Also, a new feature from the foundational viewpoint is that, in contrast with other approaches, we have uniqueness-of-types property, which is not modulo beta-conversion.

Contents

List of Figures

List of Tables

Part I

Introductory Background

1 High Level Overview

The focus of this dissertation is the algebraic data types. We propose and describe the automatic and interactive deductive systems associated with reasoning about algebraic data types, their members, and functions operating on them. This introductory background has the following purpose: (i) to give a preview of what is presented in the next parts, and (ii) to highlight and explain the shared formal components that are used throughout the whole document. From the very beginning it will be useful to frame our entire work in terms of logical signatures and associated typed deductive systems.

The Objectives of this Thesis can be summarized as follows. For the fully automatic system, our goal is composed of the following sub-goals:

- to carry out the construction in many-sorted logic, for multi-constructor types
- to allow for arbitrary well-founded mutual recursion between the algebraic data types
- to present the decision procedure abstractly and delegate the concretization to the implementation phase
- to allow for customized strategies within the framework that can yield high practical performance

In the case of an *interactive system*, our motivations are as follows:

- to build a point-and-click user interface that is easy to learn and use
- to implement the entire system based on the Curry-Howard Isomorphism
- to adapt the underlying type system and allow for the absence of the conversion rule

The whole document is comprised of four main parts and two additional parts. Below we give a general overview of each part. The related work is described within the context of the respective parts. Following this introductory section, we give an example of an elementary type system as a tool for deduction, and provide an introductory setting for reasoning about algebraic data structures and their semantics.

In order to provide motivation for part II, we note that, in general, application of functions in the theory of algebraic types must sometimes be undefined. Therefore, an adequate theory to that effect requires dealing with partial functions.

Part II: Semantics of Partial Functions. Part II proposes a typed formalism of the first order logic with partial functions, and gives the method of handling the semantics of partial functions. Most approaches to automated deduction assume a mathematical formalism in which functions are total, even though partial functions occur naturally in many applications. Furthermore, although there have been various proposals for logics of partial functions, there is no consensus on which is "the right" logic to use for verification applications. In part II we propose using a three-valued Kleene logic, where partial functions return the "undefined" value when applied outside of their domains. The particular semantics are chosen according to the *principle of least surprise* to the user; if there is disagreement among the various approaches on what the value of the formula should be, its evaluation is undefined. We show that the problem of checking validity in the three-valued logic can be reduced to checking validity in a standard two-valued logic. The typed formalism for part II is shown in Figures 2 and 3.

Part III: Deciding Theories of Algebraic Data Types. Algebraic data types are commonly used in programming. In particular, functional languages support such structures explicitly. The same notion is also a convenient abstraction for common data types such as records and linked lists or trees used in more conventional programming languages. The ability to reason automatically and efficiently about these data structures provides an important tool for the analysis and verification of programs.

Part III describes an abstract decision procedure for satisfiability of the quantifier-free formulas in the theory of algebraic data types relative to the intended model of that theory. In the past, decision procedures have been proposed for various theories of algebraic data types, some focused on the universal fragment, and some focused on handling arbitrary quantifiers. Because of the complexity of the full theory, previous work on the full theory has not focused on strategies for practical implementation. However, even for the universal fragment, previous work has been limited in several significant ways. In part III, we present a general and practical algorithm for the universal fragment. The algorithm is presented declaratively as a set of abstract rules which we show to be terminating, sound, and complete. We show how other algorithms can be realized as strategies within our general framework, and we propose a new strategy and give experimental results indicating that it performs well in practice. The typed signature for the theory of algebraic types is shown in Figures 4 and 5.

Equality Predicates. A distinctive feature of our treatment is the usage of the equality predicate. We accept the notion of equality as a built-in primitive in each of our formalisms. This is especially uncommon in the context of part IV. The equality predicates in the formal systems that we study are denoted by $\approx$, while the informal equality in the meta language of discourse is denoted by $=$ as usual. A particular usage of equality $=$ is when the two sides are syntactically identical. In this case we use the equivalence symbol $\equiv$.

Part IV: Deductive Reasoning about Algebraic Data Structures. A deductive approach to verifying assertions about algebraic structures becomes necessary when the set of formulas, valid in the intended model of the theory is undecidable. In particular, this happens in second order logic. Part IV presents a concept theorem prover, called **Term Builder**. We have developed it to support reasoning about both interpreted and uninterpreted functions and predicates over algebraic types. **Term Builder** is not limited to reasoning in an algebraic setting, and can be also used for interactive proofs in theories that admit a type-theoretic representation. The deductive system of **Term Builder** is based on dependent type theory, which subsumes standard higher order logic. Traditionally, explicit equality predicates are not necessarily supported in pure type theory. As a consequence, the uniqueness-of-types property must sometimes be sacrificed due to the introduction of the “conversion rule”. We show how to preserve uniqueness of types by replacing the conversion rule by explicit support of equality.

Part V: Using the Theorem Prover. This part describes the user interface of the prover. It explains the relationship between the prover commands and the type rules that are used to build the proof objects. In addition, it goes through an example of a proof session.

2 The Concept of an Algebraic Data Type

In this section we will give a general idea about the concept of an algebraic data type. This includes a short preview of how they are declared and used. We base this exposition on several commonly known examples. In what follows, the term "algebraic data type" may be abbreviated as ADT.

Perhaps the best-known example of a simple algebraic data type is the *list* type used in LISP. Lists are either the *null* list or are constructed from other lists using the *constructor cons*. This constructor takes two arguments and returns the result of prepending its first argument to the list in its second argument. To access the elements of a list, a pair of *selectors* is provided: *car* returns the first element of a list and *cdr* returns the rest of the list. Another simple algebraic data type is natural numbers with *zero* and *successor*. Natural numbers and lists can be captured by the following declarations. Note that the elements of lists are also taken to be lists:

$$\begin{array}{lll} \mathit{nat} & ::= & \mathit{succ}(\mathit{pred} : \mathit{nat}) \mid \mathit{zero}; \\ \mathit{list} & ::= & \mathit{cons}(\mathit{car} : \mathit{list}, \mathit{cdr} : \mathit{list}) \mid \mathit{null}; \end{array}$$

The type *nat* has two constructors: *zero*, which takes no arguments; and *succ*, which takes a single argument of type *nat* and has the corresponding selector *pred*. The primary use of ADTs is the traversal by recursive functions, which may apply and remove constructors. For example, the addition of natural numbers is defined by recursion over the second operand:

$$(+) \;=\; \lambda x : nat.\ \lambda y : nat.\ \textbf{if}\ (y \approx zero)\ \textbf{then}\ x\ \textbf{else}\ succ(x + pred(y))\ \textbf{endif};$$

It will be shown in section 5.3 that another formalization of ADTs may be carried out in Set Theory. In particular, in the following notation, μ is the set-theoretic least fixed point operator, while *succ* and *cons* are set transformers:

$$\begin{aligned} nat \quad &::= \mu N.\ succ(N) \cup \{zero\}; \\ list \quad &::= \mu L.\ cons(L, L) \cup \{null\}; \end{aligned}$$

More generally, we are interested in any set of (possibly mutually recursive) algebraic data types, each of which is built with one or more constructors. Each constructor has selectors that can be used to retrieve the original arguments as well as a *tester* which indicates whether a given term was constructed using that constructor. Consider the following mutually recursive example. The *list* type is as before, except that we now specify that the elements of the list are of type *tree*, and not *list*. The *tree* type in turn has two constructors: *node*, which takes an argument of type *list* and has the corresponding selector *children*, and *leaf*, which takes an argument of type *nat* and has the corresponding selector *data*:

$$\begin{aligned} nat \quad &:= \quad succ(pred : nat) \mid zero; \\ list \quad &:= \quad cons(car : tree,\ cdr : list) \mid null; \\ tree \quad &:= \quad node(children : list) \mid leaf(data : nat); \end{aligned}$$

The testers for this set of data types are *is_succ*, *is_zero*, *is_cons*, *is_null*, *is_node*, and *is_leaf*. Propositions about a set of inductive data types can be captured in a sorted first-order language which closely resembles the structure of the data types themselves in that it has function symbols for each constructor and selector, and a predicate symbol for each tester. For instance, propositions that we would expect to be true for the example above include the following:

1. $\forall\, x : \mathit{nat}.\ \mathit{succ}(x) \not\approx \mathit{zero}$,
2. $\forall\, x : \mathit{list}.\ x \approx \mathit{null} \vee \mathit{is\_cons}(x)$, and
3. $\forall\, x : \mathit{tree}.\ \mathit{is\_leaf}(x) \rightarrow (\mathit{data}(x) \approx \mathit{zero} \vee \mathit{is\_succ}(\mathit{data}(x)))$.

In part III of the thesis we construct a procedure for deciding such formulas. We focus on satisfiability of a set of literals, which (through well-known reductions) can be used to decide the validity of universal formulas.

3 Type Systems and Deductive Systems

3.1 The Concept of a Type System

Here we shall state our notational conventions used throughout parts I – V. For instance, $t(x)$ denotes an expression t with a possible occurence of the free variable x. As usual, $FV(t)$ denotes the set of free variables in t. Expression $t\{a/x\}$ corresponds to t after substituting a for variable x. When we write $t\{a\}$ instead, we implicitly mean the same substitution when there is only one free variable in t. Furthermore, $t_1 \equiv t_2$ denotes syntactic identity of expressions t_1, t_2. This is, of course, a much stronger property than the equality predicate ($\approx_A$), which we have given here instantiated with A as the domain of individuals. Subexpression $(a : A)$ is commonly used to indicate that a has type A. Superscript type annotations, like a^A, are useful when trying to infer the type of a subexpression. Statements of the form $\forall x : A.\ B$ express that for any a of type A, the proposition $B\{a/x\}$ holds. By Γ we usually mean a context, which is a sequence of variable declarations of the form $\{x : X,\ y : Y,\ z : Z,\ \ldots\}$. An empty context is denoted by $\emptyset$. More formally,

BNF for contexts: $\Gamma ::= \emptyset \mid \Gamma, x : A$

Type Judgements: $\Gamma \vdash a : A$, which means that a is of type A in context Γ

An important requirement for contexts is that a declared variable, (for example y), may only occur freely in the types to the right of it (in Z in this case, but not in X or Y). With this in mind, our context Γ, in general, looks like $\{x : X,\ y : Y(x),\ z : Z(x, y),\ \ldots\}$. This will be needed to support formation of dependent types in part IV.

It will be useful to consider an example of a typed deductive system or, more simply, a type system. A type system is comprised of type rules. Each rule has zero or more clauses as premisses and the conclusion clause. Each clause is a type assignment of the form $\Gamma \vdash t : T$. In Figure 1 we give an example of a basic type system $\mathcal{T}_0$. The term *sort* is usually used to denote an atomic type, while *type* is a more general term. There are two reserved sorts Type and Prop, representing the type of types and type of propositions (formulas) respectively. In the system $\mathcal{T}_0$, these sorts are treated as distinct entities, however, in part IV of our treatment they are identified. Derivations in a type system are comprised of the successive application of the type rules. An example of a derivation in $\mathcal{T}_0$ is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{X:\mathsf{Type}\vdash X:\mathsf{Type}}}{X:\mathsf{Type},x:X\vdash x:X}\qquad \dfrac{\dfrac{X:\mathsf{Type}\vdash \mathsf{Prop}:\mathsf{Type}}{X:\mathsf{Type}\vdash X\to\mathsf{Prop}:\mathsf{Type}}}{X:\mathsf{Type},Q:X\to\mathsf{Prop}\vdash Q:X\to\mathsf{Prop}}}{X:\mathsf{Type},Q:X\to\mathsf{Prop},x:X\vdash Q(x):\mathsf{Prop}}}{X:\mathsf{Type},Q:X\to\mathsf{Prop},x:X\vdash Q(x)\to Q(x):\mathsf{Prop}}}{X:\mathsf{Type},Q:X\to\mathsf{Prop}\vdash(\forall x:X.\ Q(x)\to Q(x)):\mathsf{Prop}}$$

$$\dfrac{\Gamma\vdash T:\mathsf{Type}}{\Gamma,x:T\vdash x:T}\ \text{(start)}\qquad\qquad \dfrac{\text{(axiom)}}{\Gamma,X:\mathsf{Type}\vdash X:\mathsf{Type}}$$

$$\dfrac{\Gamma,x:T_1\vdash t:T_2\quad \Gamma\vdash T_1\to T_2:\mathsf{Type}}{\Gamma\vdash(\lambda x:T_1.\ t):T_1\to T_2}\qquad\qquad \dfrac{\text{(axiom)}}{\Gamma,X:\mathsf{Prop}\vdash X:\mathsf{Prop}}$$

$$\dfrac{\Gamma\vdash\phi_1:\mathsf{Prop}\quad \Gamma\vdash\phi_2:\mathsf{Prop}}{\Gamma\vdash\phi_1\to\phi_2:\mathsf{Prop}}\qquad\qquad \dfrac{\Gamma\vdash T_1:\mathsf{Type}\quad \Gamma\vdash T_2:\mathsf{Type}}{\Gamma\vdash T_1\to T_2:\mathsf{Type}}$$

$$\dfrac{\Gamma\vdash f:T_1\to T_2\quad \Gamma\vdash t:T_1}{\Gamma\vdash f(t):T_2}\qquad\qquad \dfrac{\text{(axiom)}}{\Gamma\vdash\bot:\mathsf{Prop}}\qquad \dfrac{\text{(axiom)}}{\Gamma\vdash\mathsf{Prop}:\mathsf{Type}}$$

$$\dfrac{\Gamma\vdash P:T\to\mathsf{Prop}\quad \Gamma\vdash t:T}{\Gamma\vdash P(t):\mathsf{Prop}}\qquad\qquad \dfrac{\Gamma,x:T\vdash\phi:\mathsf{Prop}\quad \Gamma\vdash T:\mathsf{Type}}{\Gamma\vdash(\forall x:T.\ \phi):\mathsf{Prop}}$$

Figure 1: Example of a Type System

3.2 Typed Signature of the First Order Logic

In part II we use the following syntax for first order logic. Let $\Sigma=(S,\ F,\ P,\ C)$ be a signature, where $S=\{s_1,\ldots\}$ is a set of sorts, $F=\{f_1,\ldots\}$, $P=\{p_1,\ldots\}$ and $C=\{c_1,\ldots\}$ are sets of function, predicate, and constant symbols. Each symbol has a type built out of the sorts in Σ. Define a *term* t as follows:

$$t \ ::= \ x \mid c \mid f(t_1, \ldots, t_n) \mid \textbf{if } \phi \textbf{ then } t_1 \textbf{ else } t_2 \textbf{ endif},$$

where x is a variable, and the symbols c and f are from Σ, and ϕ in the conditional operator is a formula. A *formula* ϕ is defined as follows:

$$\begin{aligned} \phi \ ::= \ & \mathsf{true} \mid \mathsf{false} \mid p(t_1, \ldots, t_n) \mid t_1 \approx t_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi_1 \mid \phi_1 \wedge \phi_2 \\ & \textbf{if } \phi_0 \textbf{ then } \phi_1 \textbf{ else } \phi_2 \textbf{ endif} \mid \exists x : s.\, \phi_1 \mid \forall x : s.\, \phi_1, \end{aligned}$$

where p is a predicate from Σ. The type-theoretic formalization of signature Σ appears in Figures 2 and 3. To check that a term or formula is well-typed, we only need to apply the type system in Figure 3.

domains	$\{\mathsf{Type}, \mathsf{Prop}\}$	sorts of types and formulas respectively.
sorts	$s \in \{s_1, \ldots s_{n_s}\}$	n_s = number of atomic types (sorts).
constants	$c \in \{c_1, \ldots c_{n_c}\}$	n_c = number of constants, each c of sort s_c.
functions	$f \in \{f_1, \ldots f_{n_f}\}$	n_f = number of function symbols,
		k_f = number of arguments of f,
		$s_{(f,i)}$ = type of i-th argument of f,
		s_f = return type of f.
predicates	$p \in \{\approx_s, p_1, \ldots p_{n_p}\}$	n_p = number of predicate symbols,
		k_p = number of arguments of p,
		$s_{(p,i)}$ = type of i-th argument of p.

Figure 2: Signature for Many-Sorted First Order Logic

3.3 Background in Universal Algebra

Our notation for a typed function or a predicate symbol w in a signature Σ may be facilitated as follows. We may write $w : s_1 \cdots s_n \rightarrow s_w$ to denote that w takes n

$$\frac{}{\Gamma \vdash s_i : \mathsf{Type}} \quad (i \in \{1, \ldots, n_s\}) \qquad \frac{}{\Gamma, x : s_i \vdash x : s_i} \quad (i \in \{1, \ldots, n_s\})$$

$$\frac{\Gamma \vdash t_i : s_{(f,i)} \quad (\forall i \in \{1, \ldots, k_f\})}{\Gamma \vdash f(t_1, \ldots t_{k_f}) : s_f} \qquad \frac{\Gamma \vdash t_i : s_{(p,i)} \quad (\forall i \in \{1, \ldots, k_p\})}{\Gamma \vdash p(t_1, \ldots t_{k_p}) : \mathsf{Prop}}$$

$$\frac{\Gamma \vdash \phi : \mathsf{Prop} \quad \Gamma \vdash t_1, t_2 : s_i \ (i \in \{1, \ldots, n_s\})}{\Gamma \vdash \textbf{if } \phi \textbf{ then } t_1 \textbf{ else } t_2 \textbf{ endif} : s_i} \qquad \frac{\Gamma \vdash \phi : \mathsf{Prop}}{\Gamma \vdash \neg\phi : \mathsf{Prop}}$$

$$\frac{\Gamma \vdash \phi, \phi_1, \phi_2 : \mathsf{Prop}}{\Gamma \vdash \textbf{if } \phi \textbf{ then } \phi_1 \textbf{ else } \phi_2 \textbf{ endif} : \mathsf{Prop}} \qquad \frac{\Gamma \vdash \phi_1 : \mathsf{Prop} \quad \Gamma \vdash \phi_2 : \mathsf{Prop}}{\Gamma \vdash \phi_1 \vee \phi_2 : \mathsf{Prop}}$$

$$\frac{\Gamma, x : s_i \vdash \phi : \mathsf{Prop} \quad (i \in \{1, \ldots, n_s\})}{\Gamma \vdash \exists x : s_i .\ \phi : \mathsf{Prop}} \qquad \frac{(i \in \{1, \ldots, n_c\})}{\Gamma \vdash c_i : s_{c_i}}$$

$$\frac{\Gamma, x : s_i \vdash \phi : \mathsf{Prop} \quad (i \in \{1, \ldots, n_s\})}{\Gamma \vdash \forall x : s_i .\ \phi : \mathsf{Prop}} \qquad \frac{\Gamma \vdash \phi_1 : \mathsf{Prop} \quad \Gamma \vdash \phi_2 : \mathsf{Prop}}{\Gamma \vdash \phi_1 \wedge \phi_2 : \mathsf{Prop}}$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Prop}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Prop}} \qquad \frac{\Gamma \vdash t_1 : s_i \quad \Gamma \vdash t_2 : s_i}{\Gamma \vdash t_1 \approx t_2 : \mathsf{Prop}}$$

Figure 3: Type System for Many-Sorted First Order Logic

arguments of sorts $s_1, \ldots, s_n$ respectively, and returns a value of type s_w. In case w is a predicate, $s_w \equiv \mathsf{Prop}$. A particular model $\mathcal{M}$ of signature Σ is called a Σ-algebra. It consists of the semantic interpretation $\mathcal{M}(s)$ and $\mathcal{M}(w)$ of each sort s by a set, and each operation w in Σ by a mapping. Our model $\mathcal{M}$ interprets each $w : s_{w,1} \cdots s_{w,k_w} \rightarrow s_w$ by a mapping

$$\mathcal{M}(w) : \mathcal{M}(s_{(w,1)}) \cdots \mathcal{M}(s_{(w,k_w)}) \rightarrow \mathcal{M}(s_w).$$

$\mathcal{M}$ also interprets ground terms in Σ. For each $t : s$, the semantics of the term t in the model $\mathcal{M}$ is $[\![t]\!]_{\mathcal{M}} \in \mathcal{M}(s)$. It is defined inductively as follows:

$$[\![w(t_1, \ldots, t_{k_w})]\!]_{\mathcal{M}} = \mathcal{M}(w)([\![t_1]\!]_{\mathcal{M}}, \ldots, [\![t_{k_w}]\!]_{\mathcal{M}}).$$

In case we deal with a nullary symbol $w : s_w$, we write: $[\![w]\!]_{\mathcal{M}} = \mathcal{M}(w)$. Suppose a context Γ is declared. A variable assignment $e : \Gamma \rightarrow \mathcal{M}$ is a function that assigns to each x of sort s from Γ a value $a \in \mathcal{M}(s)$. An augmentation of a variable assignment e is denoted by $e(x \leftarrow a)$, where $\Gamma \vdash x : s$, and $a \in \mathcal{M}(s)$. A notion of homomorphisms is associated with Σ-algebras via the following definition.

Definition 3.1. *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two Σ-algebras, and let $h : \mathcal{M}_1 \to \mathcal{M}_2$ be a family of mappings $\{h_s : \mathcal{M}_1(s) \to \mathcal{M}_2(s) \mid s \text{ is a sort in } \Sigma\}$, such that for any signature symbol $w : s_{(w,1)} \cdots s_{(w,n_w)} \to s_w$, the following holds:*

$$h_{s_w}(\mathcal{M}_1(w)(a_1, \dots a_{k_w}))) = \mathcal{M}_2(w)(h_{s_{(w,1)}}(a_1), \dots, h_{s_{(w,k_w)}}(a_{k_w}))$$

where each $a_i \in \mathcal{M}_1(s_{(w,i)})$. Then h is a homomorphism from $\mathcal{M}_1$ to $\mathcal{M}_2$.

Another algebraic notion is a congruence relation $\equiv_h$ induced by a homomorphism h. For two elements $a_1, a_2 \in \mathcal{M}_1(s)$, $a_1 \equiv_h a_2$ iff $h_s(a_1) = h_s(a_2)$. This means that a_1 and a_2 are in the same equivalence class of $\equiv_h$ iff their h-images coincide. If h is both surjective and injective, it is called an isomorphism, since in this case the inverse map h^{-1} also exists and is a homomorphism. Two Σ-algebras are isomorphic, iff there is an isomorphism between them. In conjunction with any congruence relation $\equiv_Q$ for a Σ-algebra $\mathcal{M}$ there is a notion of a quotient algebra $\mathcal{M}/\equiv_Q$. This algebra operates on the equivalence classes of $\equiv_Q$ rather than on elements on $\mathcal{M}$. For any operation symbol w in Σ:

$$\mathcal{M}/\equiv_Q (w)([a_1]_{\equiv_Q}, \dots, [a_{k_w}]_{\equiv_Q}) = [\mathcal{M}(w)(a_1, \dots, a_{k_w})]_{\equiv_Q}$$

where $a_i \in \mathcal{M}(s_{(w,i)})$ and $[a]_{\equiv_Q}$ is an equivalence class of $\equiv_Q$. The quotient algebra $\mathcal{M}/\equiv_Q$ is well defined, namely, for all operation symbols w in Σ, and any $a_i, b_i \in \mathcal{M}(s_{(w,i)})$ the following condition holds:

$$(\forall i \in \{1, \dots, k_w\}.\ a_i \equiv_Q b_i) \ \ \textit{implies} \ \ \mathcal{M}(w)(a_1, \dots, a_{k_w}) \equiv_Q \mathcal{M}(w)(b_1, \dots, b_{k_w})$$

Theorem 3.2. *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two Σ-algebras, and let $h : \mathcal{M}_1 \rightarrow \mathcal{M}_2$ be a surjective homomorphism. Then the quotient algebra $\mathcal{M}_1/\equiv_h$ is isomorphic to $\mathcal{M}_2$.*

Proof. For a treatment of Σ-algebras, congruence relations, and homomorphisms, see reference [28]. The proof of this theorem appears in [28], page 52, Theorem 3.21. ☐

4 Theory of Algebraic Data Types

4.1 Signature for the Theory of Algebraic Types

We formalize ADTs in the context of many-sorted equational logic (see [33] or [28] among others). This is subsumed by the typed first order logic with equality. We assume a many-sorted signature Σ whose set of sorts consists of a distinguished sort Prop for propositions, and $p \geq 1$ sorts $\tau_1, \ldots, \tau_p$ for the ADTs. We will denote by s, possibly with subscripts, any sort in the signature other than Type, by τ any sort in $\{\tau_1, \ldots, \tau_p\}$.

The function symbols in our theory signature correspond to the constructors, selectors, and testers of the set of ADTs under consideration. We assume for each τ a set $\mathcal{C}_\tau$ of $m_\tau \geq 1$ *constructors* of τ. We will denote constructors by the letter C, possibly primed or with subscripts. We will write $C : s_1 \cdots s_n \rightarrow \tau$ to denote that the constructor C takes $n \geq 0$ arguments of respective sorts $s_1, \ldots, s_n$ and returns a value of sort τ. Constructors with arity 0 are called *nullary constructors* or *constants*. For each constructor $C : s_1 \cdots s_n \rightarrow \tau$, we assume n corresponding *selector* symbols denoted by $S_C^{(1)}, \ldots, S_C^{(n)}$ with $S_C^{(i)} : \tau \rightarrow s_i$, and a *tester* predicate symbol denoted by is_C of type $\tau \rightarrow \mathsf{Prop}$. We write $S^{(i)}$ instead of $S_C^{(i)}$ when C is

domains	$\{\mathsf{Type}, \mathsf{Prop}\}$ (sorts of types and formulas respectively)
base propositions	$\{\mathsf{true} : \mathsf{Prop},\ \mathsf{false} : \mathsf{Prop}\}$
algebraic types	$\{\tau_i : \mathsf{Type}\}_{i=1}^{p}$
constructors	$\{C^i_\tau : s_{(C^i_\tau,1)} \cdots s_{(C^i_\tau,k_{C^i_\tau})} \to \tau\}_{i=1}^{m_\tau}$
selectors	$\{S^{(j)}_C : s_C \to s_{(C,j)}\}_{j=1}^{k_C}$, where C is a constructor
equality	$\{\approx\ :\ \tau_i \times \tau_i \to \mathsf{Prop}\}_{i=1}^{p}$, and $\approx: \mathsf{Prop} \times \mathsf{Prop} \to \mathsf{Prop}$
testers	$\{is_C : s_C \to \mathsf{Prop}\}$, where C is a constructor

p	number of algebraic types τ_i
m_τ	number of constructors of the type τ
C^i_τ	i-th constructor of type τ
k_C	number of arguments of constructor C
$S^{(j)}_C$	selector of the j-th component from terms of constructor C
$s_{(C,i)}$	type of the i-th argument of constructor C
s_C	return type of constructor C

Figure 4: Signature Σ for the Theory of Algebraic Data Types

clear from context or not important.

In addition to these symbols, we also assume that the signature contains two constants, true and false of sort Prop. As usual in many-sorted equational logic, we also have $p+1$ equality symbols (one for each sort mentioned above), all written as $\approx$.

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Prop}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Prop}}$$

$$\frac{i \in \{1 \ldots p\}}{\Gamma \vdash \tau_i : \mathsf{Type}} \qquad \frac{i \in \{1 \ldots p\}}{\Gamma, x : \tau_i \vdash x : \tau_i}$$

$$\frac{\Gamma \vdash t_i : s_{(C,i)} \quad i \in \{1, \ldots, k_C\}}{\Gamma \vdash C(t_1, \ldots t_{k_C}) : s_C} \qquad \frac{\Gamma \vdash t : s_C \quad j \in \{1, \ldots, k_C\}}{\Gamma \vdash S^{(j)}_C(t) : s_{(C,j)}}$$

$$\frac{\Gamma \vdash t_1 : \tau_i \quad \Gamma \vdash t_2 : \tau_i \quad i \in \{1, \ldots, p\}}{\Gamma \vdash (t_1 \approx t_2) : \mathsf{Prop}} \qquad \frac{\Gamma \vdash t : s_C \quad s_C = s_{C'} \in \{\tau_1, \ldots, \tau_p\}}{\Gamma \vdash is_{C'}(t) : \mathsf{Prop}}$$

Figure 5: Type System for the Theory of ADTs

4.2 Axiomatization by Equational Specification

Members of algebraic types are the structures over which the logical and programmic branching can be implemented. Each declared algebraic type has a set of constructors, based on which case analysis is performed. Each constructor carries a vector of data items, each of its own algebraic type. In the following, assume an algebraic type τ is defined by this grammar:

$$\tau ::= \{C_1\ X_{1,1}\ X_{1,2}\ ...\ X_{1,k_1} \mid ... \mid C_n\ X_{n,1}\ X_{n,2}\ ...\ X_{n,k_n}\};$$

where C_i are constructors, and each $X_{i,j}$ is the sort of the j-th argument of the constructor C_i. Let $(x_{(i,j)} : X_{i,j})$ and $(v_{(i,j)} : X_{i,j})$. An element of type τ is denoted in the following way:

$$C_i(v_{(i,1)},\ v_{(i,2)},\ ...\ v_{(i,k_i)}), \text{ or equivalently, } (C_i\mathbf{v_i}).$$

Here the vector of v-values $\mathbf{v_i}$ is the data stored with the constructor C_i. In case we deal with variable placeholders, we denote this as follows:

$$C_i(x_{(i,1)},\ x_{(i,2)},\ ...\ x_{(i,k_i)}), \text{ or equivalently, } (C_i\mathbf{x_i}).$$

Substitution of values for the variables in an expression M is denoted:

$$M\{v_{(i,j)}/x_{(i,j)}\}_{j=1}^{k_i}, \text{ or equivalently, } M\{\mathbf{v_i}/\mathbf{x_i}\}.$$

Our theory of ADTs also requires that all data types are well-founded. This will be explained further in part III. Previous work on algebraic data types [50, 51] uses first-order axiomatizations in an attempt to capture the main properties of a data type and reason about it. We find it simpler and cleaner to use a semantic approach instead, as is done in algebraic specification. A set of ADTs can be given a simple equational specification over a suitable signature. The intended model for our theory can be formally, and uniquely, defined as the initial model of this specification. Reasoning about a set of ADTs then amounts to reasoning about formulas that are true in this particular initial model.

Given the signature Σ, the associated algebraic types are specified by the following set $\mathcal{E}$ of axiom schemas for each ADT τ in Σ and distinct constructors $C : s_1 \cdots s_n \to \tau$ and $C' : s'_1 \cdots s'_{n'} \to \tau$. These axioms are formulated to capture the intended model of the theory of ADTs. As we shall see, this model is freely generated by constructor terms.

$$
\begin{array}{ll}
\forall x_1, \ldots, x_n.\ is_C(C(x_1, \ldots, x_n)) \approx \mathsf{true} & \\
\forall x_1, \ldots, x_n.\ is_{C'}(C(x_1, \ldots, x_n)) \approx \mathsf{false} & \\
\forall x_1, \ldots, x_n.\ S_C^{(i)}(C(x_1, \ldots, x_n)) \approx x_i & \text{for all } i = 1, \ldots, n \\
\forall x_1, \ldots, x_n.\ S_{C'}^{(i)}(C(x_1, \ldots, x_n)) \approx t_{C'}^{i} & \text{for all } i = 1, \ldots, n'
\end{array}
$$

Note the situation when the selector $S_{C'}^{(i)}$ is applied to the term constructed with C, where $C' \neq C$. Our axiom specifies that in this case, the result is some designated ground term $t_{C'}^{i}$ of type $s_{(C',i)}$. Reference [35], sections 3.6.3 and 3.6.4 advocate a similar approach to undefindedness. This is different from other treatments (such as [22, 50, 51]) where the application of a selector to the wrong constructor is treated as the identity function. The main reason for this difference is that the identity function would not always be well-typed in many-sorted logic. In part II this feature of our treatment is particularly relevant, as it will be necessary to extend the natural partial model of an ADT to an arbitrarily chosen total model.

5 Algebraic Types as Term Algebras

5.1 Term Algebras as Models

Assume that our theory of ADTs is specified using signature Σ. We can inductively define the semantics $[\![\ ]\!]_{\mathcal{T}(\Sigma)}$ of terms in Σ as the term algebra $\mathcal{T}(\Sigma)$. We may sometimes use $\mathcal{T}(\Sigma)$ to denote just the set of terms of the signature Σ, instead of their term algebra.

Definition 5.1. *freely generated term algebra $\mathcal{T}(\Sigma)$*

$$
\begin{array}{lcl}
[\![C(t_1,\ldots,t_{k_C})]\!]_{\mathcal{T}(\Sigma)} & = & C([\![t_1]\!]_{\mathcal{T}(\Sigma)},\ldots[\![t_{k_C}]\!]_{\mathcal{T}(\Sigma)}) \\
[\![S_C^{(i)}(t)]\!]_{\mathcal{T}(\Sigma)} & = & S_C^{(i)}([\![t]\!]_{\mathcal{T}(\Sigma)}) \\
[\![is_C(t)]\!]_{\mathcal{T}(\Sigma)} & = & is_C([\![t]\!]_{\mathcal{T}(\Sigma)}) \\
[\![\mathsf{true}]\!]_{\mathcal{T}(\Sigma)} & = & \mathsf{true} \\
[\![\mathsf{false}]\!]_{\mathcal{T}(\Sigma)} & = & \mathsf{false}
\end{array}
$$

Now let Ω be the signature obtained from Σ by removing selectors and testers. We can also inductively define the semantics $[\![\ ]\!]_{\mathcal{T}(\Omega)}$ of terms in Σ as the term algebra $\mathcal{T}(\Omega)$. We may sometimes use $\mathcal{T}(\Omega)$ to denote just the set of terms of the signature Ω, instead of their term algebra.

Definition 5.2. *constructor generated term algebra $\mathcal{T}(\Omega)$*

$$
\begin{array}{lcl}
[\![C(t_1,\ldots,t_{k_C})]\!]_{\mathcal{T}(\Omega)} & = & C([\![t_1]\!]_{\mathcal{T}(\Omega)},\ldots[\![t_{k_C}]\!]_{\mathcal{T}(\Omega)}) \\
[\![S_C^{(i)}(t)]\!]_{\mathcal{T}(\Omega)} & = & \textit{if } [\![t]\!]_{\mathcal{T}(\Omega)} = C(t_1,\ldots,t_{k_C}) \textit{ then } t_i \textit{ else } t_C^i \\
[\![is_C(t)]\!]_{\mathcal{T}(\Omega)} & = & \textit{if } [\![t]\!]_{\mathcal{T}(\Omega)} = C(t_1,\ldots,t_{k_C}) \textit{ then } \mathsf{true} \textit{ else } \mathsf{false} \\
[\![\mathsf{true}]\!]_{\mathcal{T}(\Omega)} & = & \mathsf{true} \\
[\![\mathsf{false}]\!]_{\mathcal{T}(\Omega)} & = & \mathsf{false}
\end{array}
$$

The following lemma clarifies the correctness of the definition of $[\![\ ]\!]_{\mathcal{T}(\Omega)}$:

Lemma 5.3. *If $[\![t]\!]_{\mathcal{T}(\Omega)} = C(t_1,\ldots,t_{k_C})$ then each $t_i \equiv [\![t_i]\!]_{\mathcal{T}(\Omega)}$.*

Proof. This lemma can be proved by a standard inductive argument from Definition 5.2. □

Informally stated, this lemma expresses the fact that the term-algebraic semantics $[\![\ ]\!]_{\mathcal{T}(\Omega)}$ is comprised of terms which are structurally made of constructor symbols. This is in contrast with the term-algebraic semantics $[\![\ ]\!]_{\mathcal{T}(\Sigma)}$ that preserves the structure of terms. The following definition will clarify the precise relation between these two term algebras.

Definition 5.4. *Let $h : \mathcal{T}(\Sigma) \to \mathcal{T}(\Omega)$ be a family of maps h_s for each sort s in Σ:*

$$\textit{for each term } t \in \mathcal{T}(\Sigma) \textit{ of sort } s\textit{: } h_s([\![t]\!]_{\mathcal{T}(\Sigma)}) = [\![t]\!]_{\mathcal{T}(\Omega)}$$

Lemma 5.5. *Mapping h from Definition 5.4 is a homomorphism from $\mathcal{T}(\Sigma)$ to $\mathcal{T}(\Omega)$.*

Proof. This result can be shown by standard structural induction over the term algebra $\mathcal{T}(\Sigma)$ using Lemma 5.3 and Definitions 5.1, 5.2. □

5.2 Standard Results from Universal Algebra

By standard results in universal algebra we know that $\mathcal{E}$ admits an *initial model* $\mathcal{R}$. We refer the reader to [33] for a thorough treatment of initial models. For our purposes, it will be enough to mention the following properties that $\mathcal{R}$ enjoys by virtue of being an initial model.

Lemma 5.6. *Where $\approx_{\mathcal{E}}$ is the equivalence relation on Σ-terms induced by $\mathcal{E}$, let $\mathcal{T}(\Sigma)/\approx_{\mathcal{E}}$ be the quotient of the term algebra $\mathcal{T}(\Sigma)$ by $\approx_{\mathcal{E}}$.*

1. *For all ground Σ-terms t_1, t_2 of the same type, $t_1 \approx_{\mathcal{E}} t_2$ iff $\mathcal{R}$ satisfies $t_1 \approx t_2$.*

2. *$\mathcal{R}$ is isomorphic to $\mathcal{T}(\Sigma)/\approx_{\mathcal{E}}$.*

Proof. These are applications to $\mathcal{R}$ of standard results about initial models. See, for instance Theorem 5.2.11 and Theorem 5.2.17 of [33]. $\square$

Lemma 5.7. *The model $\mathcal{R}$ is isomorphic to $\mathcal{T}(\Omega)$.*

Proof. By Lemma 5.6(2) we can take $\mathcal{R}$ to coincide with $\mathcal{T}(\Sigma)/{\approx_{\mathcal{E}}}$, whose elements are the equivalence classes of $\approx_{\mathcal{E}}$ on the ground Σ-terms. We also know that $h : \mathcal{T}(\Sigma) \to \mathcal{T}(\Omega)$ from Lemma 5.5 is surjective, since it behaves as an identity over $\mathcal{T}(\Omega)$. It follows by Theorem 3.2, that $\mathcal{T}(\Sigma)/\equiv_h$ is isomorphic to $\mathcal{T}(\Omega)$. On the other hand, it is easy to verify that $\equiv_h$ and $\approx_{\mathcal{E}}$ are the same. Hence, $\mathcal{T}(\Sigma)/{\approx_{\mathcal{E}}}$ is isomorphic to $\mathcal{T}(\Omega)$. $\square$

The claim shows that:

(i) every ground Σ-term is equivalent in $\mathcal{E}$ to a ground Ω-term.

(ii) no two distinct ground Ω-terms belong to the same equivalence class.

We will call *ground constructor terms* the elements of the set $\mathcal{T}(\Omega)$ defined in the previous lemma. Informally, the lemma means that $\mathcal{R}$ does in fact capture the set of ADTs in question, as we can take the carrier of $\mathcal{R}$ to be the term algebra $\mathcal{T}(\Omega)$. This also shows that in $\mathcal{R}$ each data type τ is generated using just its constructors, and that distinct ground constructor terms of type τ are distinct elements of the data type.

5.3 Algebraic Types as Least Fixed Points

Let $\widetilde{C} : 2^{\mathcal{T}(\Sigma)} \to 2^{\mathcal{T}(\Sigma)}$ be a mapping over the subsets of the term algebra $\mathcal{T}(\Sigma)$. In particular, it maps a set of terms X into the set that contains all possible type-correct applications of constructors in Σ to the elements of X. Let us define the mapping

$$\Phi(X) = X \bigcup \widetilde{C}(X)$$

For example, $\Phi(\emptyset) = \mathcal{B}$, the set of all nullary constructors in Σ. The following lemma follows immediately from the definition:

Lemma 5.8. *The following properties hold for the mapping* Φ*:*

- Φ *is cumulative:* $X \subseteq \Phi(X)$*.*
- Φ *is monotone: if* $X \subseteq Y$ *then* $\Phi(X) \subseteq \Phi(Y)$*.*
- Φ *is pointwise:* $\Phi(X) = \bigcup_{Y \subseteq X} \Phi(Y)$*, where each* Y *is finite.*

Lemma 5.9. Φ *is continuous: for a chain of sets* $X_0 \subseteq X_1 \subseteq X_2 \subseteq \dots$

$$\bigcup_{i=0}^{\infty} \Phi(X_i) = \Phi(\bigcup_{i=0}^{\infty} X_i)$$

Proof. First we prove the inclusion ($\subseteq$). Since every $X_i \subseteq \bigcup_{i=0}^{\infty} X_i$, then by monotonicity:

$$\Phi(X_i) \subseteq \Phi(\bigcup_{i=0}^{\infty} X_i)$$

for each i, and therefore:

$$\bigcup_{i=0}^{\infty} \Phi(X_i) \subseteq \Phi(\bigcup_{i=0}^{\infty} X_i)$$

Now we prove the opposite inclusion ($\supseteq$). For any finite $Y \subset \bigcup_{i=0}^{\infty} X_i$, let $k(Y)$ be the least index, such that $Y \subset X_{k(Y)}$. Since every Y is finite, it is subsumed by some X_i. Let $K = \{i \in \mathcal{N} \mid i = k(Y) \text{ for some } Y\}$. Then by Lemma 5.8, for finite sets Y:

$$\Phi(\bigcup_{i=0}^{\infty} X_i) = \bigcup_{Y \subset \bigcup_{i=0}^{\infty} X_i} \Phi(Y) = \bigcup_{i \in K} \bigcup_{Y \subset X_i} \Phi(Y) \subseteq \bigcup_{i=0}^{\infty} \bigcup_{Y \subset X_i} \Phi(Y) \subseteq \bigcup_{i=0}^{\infty} \Phi(X_i)$$

□

We now prove another lemma that will yield an alternative view of the intended semantics of an ADT. Namely, the least fixed point of Φ is exactly the set of terms in the intended model of Σ.

Lemma 5.10. *Let* $\Phi^0(X) = \emptyset$ *and* $\Phi^{i+1}(X) = \Phi(\Phi^i(X))$*. Let* $X_0 = \bigcup_{i=0}^{\infty} \Phi^i(\emptyset)$*. Then* $X_0 = \Phi(X_0)$ *is the least fixed point of* Φ*.*

Proof. We first show that X_0 is a fixed point of Φ. By Lemma 5.9

$$\Phi(X_0) = \Phi(\bigcup_{i=0}^{\infty} \Phi^i(\emptyset)) = \bigcup_{i=0}^{\infty} \Phi(\Phi^i(\emptyset)) = \bigcup_{i=1}^{\infty} \Phi^i(\emptyset) = X_0$$

Now suppose that $X_1 = \Phi(X_1)$. Since $\emptyset \subseteq X_1$, by monotonicity we have: $\Phi(\emptyset) \subseteq \Phi(X_1) = X_1$. By successive applications: $\Phi^i(\emptyset) \subseteq \Phi^i(X_1) = X_1$ for any index i. Therefore:

$$X_0 = \bigcup_{i=1}^{\infty} \Phi^i(\emptyset) \subseteq \bigcup_{i=1}^{\infty} \Phi^i(X_1) = X_1$$

Hence, the fixed point X_0 is subsumed by any other fixed point X_1 of Φ. □

The intended semantics of the signature Σ for the theory of ADTs is the term algebra $\mathcal{T}(\Omega)$, where Ω consists only of constructors. Testers and selectors are not part of the term model as they are *operational*, that is, an invasive programmic action is associated with them. We can infer from Lemma 5.10 that $\mathcal{T}(\Omega) = \bigcup_{i=0}^{\infty} \Phi(\emptyset)$, where $\Phi(X) = X \cup \widetilde{C}(X)$, and:

$$\widetilde{C}(X) = \{t \in \mathcal{T}(\Sigma) \mid t = C(t_1, \ldots, t_{k_C}),\ t_i \in X,\ \emptyset \vdash t : \tau,\ C \in \mathcal{C}_\tau\}$$

A result of the same semantic significance was claimed in Lemma 5.7.

Part II

Semantics of Partial Functions

6 Introduction and Related Work

This part is devoted to the formal treatment of partial functions and predicates. Although it is generally agreed that a logic which can accommodate partial functions is useful for a wide variety of applications, there is general disagreement on which logic should be used. An overview of the different approaches can be found in [17, 24]. Of the approaches which take partiality seriously as opposed to attempting a work-around, there are two main alternatives. The first allows terms to be undefined, but requires that all formulas be either true or false. The unusual feature of this approach is that a predicate applied to an undefined term is defined to be false. Although this logic preserves some nice features of classical logic (the deduction theorem, for instance), in a certain sense there is a loss of information because the undefinedness does not propagate to formulas. For example, if we assume the term $1/0$ is undefined, then the formula $\neg P(1/0)$ will be valid.

The second approach is based on Kleene's strong three-valued logic [25], and allows both terms and formulas to be undefined. This approach is more conservative in the sense that any formula which is valid in the second approach will be valid in the first approach, but there are some formulas, such as $\neg P(1/0)$, which may be valid in the first approach but will be undefined in the second.

We prefer the second approach based on a *principle of least surprise.* That is, a formula should be valid only when there is no disagreement on whether that is a reasonable conclusion. This is particularly important in verification applications, as the integrity of a system may be judged by whether a theorem about the system is valid. Furthermore, it is our experience that any theorem which really should be valid can be formulated in such a way that it is valid according to this second approach.

A more pragmatic issue that must be dealt with is that most theorem-provers are based on classical logic. Various approaches have been advocated for modifying standard theorem-proving to accommodate logics with partial functions [23, 24, 29, 46]. However, we are interested in finding a method for supporting partiality without modifying the theorem prover. One way to do this is by building over- and under-approximations for the formula. This technique has been successfully applied for three-valued model-checking [12, 21].

PVS (Prototype Verification System [42]) uses a completely different approach which involves constructing and proving additional formulas called *type correctness conditions* (TCCs). The validity of TCCs guarantees that all the relevant terms and formulas are always defined. However, TCCs in PVS can yield surprising results. For example, it is possible to have a formula of the form $A \rightarrow B$ with a valid TCC whose contrapositive $\neg B \rightarrow \neg A$ has an invalid TCC.

We propose a technique for checking the validity of a formula in three-valued logic by reducing the problem to checking two formulas in standard two-valued logic. Similarly to PVS, we construct a TCC formula whose validity implies that the original formula is always defined. After checking the TCC, we check the original formula. Both of these checks can be done using standard two-valued logic. Note that, unlike in PVS, our method is *precise* in the sense that if a TCC is invalid, the validity of the original formula is indeed undefined in the three-valued semantics.

The following sections are organized as follows. Section 7 gives the syntax and semantics for our three-valued logic. Section 8 gives two fundamental theorems which justify the reduction to two-valued logic.

7 Three-Valued Logic: Syntax and Semantics

The signature Σ that we are going use in part II, and its type-theoretic formalization, has been introduced in subsection 3.2.

It is important to distinguish the two versions of the if-then-else operator: the one for terms, and the other for formulas. Also note that the if-then-else operators are not expressible in terms of other operators or logical connectives in 3-valued logic.[1]

For our purposes, we will assume that included with every signature Σ is a set Δ of *domain formulas*, one for each function and predicate symbol in Σ. The domain formula for a function symbol f is a Σ-formula with k free variables where k is the arity of f and is denoted $\delta_f[x_1, \ldots, x_k]$. The domain formula for a predicate symbol p of arity k is defined similarly and is denoted $\delta_p[x_1, \ldots, x_k]$. An instantiation of a domain formula δ_f with terms $t_1, \ldots, t_k$ is written $\delta_f[t_1, \ldots, t_k]$ and denotes the result of replacing each x_i with t_i in the domain formula $\delta_f[x_1, \ldots, x_k]$.

Intuitively, the domain formula for f defines the set of points where f is defined. Note that our approach assumes this set is always first-order definable. Fortunately, for the practical cases we consider, this is always the case. In order to have an unambiguous semantics, it is important that the domain formulas themselves always be defined. One simple way to ensure this is to require that if s is a function or predicate symbol appearing in a domain formula, then $\delta_s[x_1, \ldots, x_n] = \mathsf{true}$.

7.1 Three-valued semantics with partial functions

Given a signature Σ, a model is defined as in Section 3.3. Namely, it maps each sort s in Σ into its carrier set $\mathcal{M}(s)$, and also gives an *interpretation*, which is a mapping

[1]The obvious 2-valued translations $(\phi_0 \to \phi_1) \wedge (\neg\phi_0 \to \phi_2)$ and $(\phi_0 \wedge \phi_1) \vee (\neg\phi_0 \wedge \phi_2)$ are actually over- and under-approximations of the 3-valued operator **if** ϕ_0 **then** ϕ_1 **else** ϕ_2 **endif**.

from constant symbols $c : s$, function symbols $f : s_1 \cdots s_n \to s$, and predicate symbols $p : s_1 \cdots s_n \to \mathsf{Prop}$ in Σ to elements $\mathcal{M}(c) \in \mathcal{M}(s)$, partial functions $\mathcal{M}(f) : \mathcal{M}(s_1) \cdots \mathcal{M}(s_n) \to \mathcal{M}(s)$, and relations $\mathcal{M}(p) \subseteq \mathcal{M}(s_1) \times \cdots \times \mathcal{M}(s_n)$, respectively.

Given a model $\mathcal{M}$ and a variable assignment e which maps each variable x of type s to an element of $\mathcal{M}(s)$, the value of an expression (a term or a formula) α is denoted $[\![\alpha]\!]_{\mathcal{M}} e$ and is defined in Figure 1. The value of a term may be an element of some $\mathcal{M}(s)$ or a distinguished value $\bot_t$ not in any $\mathcal{M}(s)$. The value of a formula may be *true*, *false*, or $\bot_\phi$. We will use $\bot$ to represent both $\bot_t$ and $\bot_\phi$ since terms and formulas are always syntactically separated from each other, and the particular kind of $\bot$ is always clear from the context.

A model is required to satisfy the following additional condition imposed by the domain formulas Δ:

$$[\![\delta_f[x_1, \ldots, x_k]]\!]_{\mathcal{M}} e = \mathit{true} \quad \text{iff} \quad \mathcal{M}(f) \text{ is defined at } ([\![x_1]\!]_{\mathcal{M}} e, \ldots, [\![x_k]\!]_{\mathcal{M}} e).$$

We say that two expressions α and β are logically equivalent, and write $\alpha \approxeq \beta$ if $[\![\alpha]\!]_{\mathcal{M}} e = [\![\beta]\!]_{\mathcal{M}} e$ for every model $\mathcal{M}$ and variable assignment e.

7.2 Semantics of if-then-else

Notice that the interpretation of the if-then-else operator (for terms) is undefined if the condition is undefined, even if the other two children evaluate to the same value. One reason for this choice of the semantics is simply that it turns out to be practical in real applications. In real programs, if a partial function is applied to an argument outside of its domain, the program may crash or raise an exception; in other words, it results in an abnormal behavior. Therefore, detecting a possible $\bot$

$$[\![c]\!]_{\mathcal{M}}e = \mathcal{M}(c), \quad [\![x]\!]_{\mathcal{M}}e = e(x), \quad [\![\mathsf{true}]\!]_{\mathcal{M}}e = \mathit{true}, \quad [\![\mathsf{false}]\!]_{\mathcal{M}}e = \mathit{false}$$

$$[\![f(t_1,\ldots,t_n)]\!]_{\mathcal{M}}e = \begin{cases} \mathcal{M}(f)([\![t_1]\!]_{\mathcal{M}}e,\ldots,[\![t_n]\!]_{\mathcal{M}}e), & \\ \quad \text{if } [\![t_i]\!]_{\mathcal{M}}e \neq \bot \text{ for all } i \in [1..n] & \\ \quad \text{and } [\![\delta_f[t_1,\ldots,t_n]]\!]_{\mathcal{M}}e = \mathit{true}; & \\ \bot \quad \text{otherwise.} & \end{cases}$$

$$[\![\ \textbf{if } \phi \textbf{ then } t_1 \textbf{ else } t_2 \textbf{ endif}\ ]\!]_{\mathcal{M}}e = \begin{cases} \bot, & \text{if } [\![\phi]\!]_{\mathcal{M}}e = \bot; \\ [\![t_1]\!]_{\mathcal{M}}e, & \text{if } [\![\phi]\!]_{\mathcal{M}}e = \mathit{true}; \\ [\![t_2]\!]_{\mathcal{M}}e, & \text{if } [\![\phi]\!]_{\mathcal{M}}e = \mathit{false}. \end{cases}$$

$$[\![\ \textbf{if } \phi \textbf{ then } \phi_1 \textbf{ else } \phi_2 \textbf{ endif}\ ]\!]_{\mathcal{M}}e = \begin{cases} \bot, & \text{if } [\![\phi]\!]_{\mathcal{M}}e = \bot; \\ [\![\phi_1]\!]_{\mathcal{M}}e, & \text{if } [\![\phi]\!]_{\mathcal{M}}e = \mathit{true}; \\ [\![\phi_2]\!]_{\mathcal{M}}e, & \text{if } [\![\phi]\!]_{\mathcal{M}}e = \mathit{false}. \end{cases}$$

$$[\![p(t_1,\ldots,t_n)]\!]_{\mathcal{M}}e = \begin{cases} \mathcal{M}(p)([\![t_1]\!]_{\mathcal{M}}e,\ldots,[\![t_n]\!]_{\mathcal{M}}e), & \\ \quad \text{if } [\![t_i]\!]_{\mathcal{M}}e \neq \bot \text{ for all } i \in [1..n] & \\ \quad \text{and } [\![\delta_p[t_1,\ldots,t_n]]\!]_{\mathcal{M}}e = \mathit{true}; & \\ \bot \quad \text{otherwise.} & \end{cases}$$

$$[\![t_1 \approx t_2]\!]_{\mathcal{M}}e = \begin{cases} [\![t_1]\!]_{\mathcal{M}}e = [\![t_2]\!]_{\mathcal{M}}e, & \text{if } [\![t_1]\!]_{\mathcal{M}}e \neq \bot \text{ and } [\![t_2]\!]_{\mathcal{M}}e \neq \bot; \\ \bot & \text{otherwise.} \end{cases}$$

$$[\![\phi_1 \wedge \phi_2]\!]_{\mathcal{M}}e = \begin{cases} \mathit{true}, & \text{if } [\![\phi_1]\!]_{\mathcal{M}}e = \mathit{true} \text{ and } [\![\phi_2]\!]_{\mathcal{M}}e = \mathit{true}; \\ \mathit{false} & \text{if } [\![\phi_1]\!]_{\mathcal{M}}e = \mathit{false} \text{ or } [\![\phi_2]\!]_{\mathcal{M}}e = \mathit{false}; \\ \bot & \text{otherwise.} \end{cases}$$

$$[\![\phi_1 \vee \phi_2]\!]_{\mathcal{M}}e = \begin{cases} \mathit{true}, & \text{if } [\![\phi_1]\!]_{\mathcal{M}}e = \mathit{true} \text{ or } [\![\phi_2]\!]_{\mathcal{M}}e = \mathit{true}; \\ \mathit{false} & \text{if } [\![\phi_1]\!]_{\mathcal{M}}e = \mathit{false} \text{ and } [\![\phi_2]\!]_{\mathcal{M}}e = \mathit{false}; \\ \bot & \text{otherwise.} \end{cases}$$

$$[\![\neg\phi]\!]_{\mathcal{M}}e = \begin{cases} \mathit{true}, & \text{if } [\![\phi]\!]_{\mathcal{M}}e = \mathit{false}; \\ \mathit{false} & \text{if } [\![\phi]\!]_{\mathcal{M}}e = \mathit{true}; \\ \bot & \text{if } [\![\phi]\!]_{\mathcal{M}}e = \bot. \end{cases}$$

$$[\![\forall x : s.\, \phi]\!]_{\mathcal{M}}e = \begin{cases} \mathit{true}, & \text{if for all } a \in \mathcal{M}(s)\text{: } [\![\phi]\!]_{\mathcal{M}}e(x \leftarrow a) = \mathit{true}; \\ \mathit{false}, & \text{if for some } a \in \mathcal{M}(s)\text{: } [\![\phi]\!]_{\mathcal{M}}e(x \leftarrow a) = \mathit{false}; \\ \bot & \text{otherwise.} \end{cases}$$

$$[\![\exists x : s.\, \phi]\!]_{\mathcal{M}}e = \begin{cases} \mathit{true}, & \text{if for some } a \in \mathcal{M}(s)\text{: } [\![\phi]\!]_{\mathcal{M}}e(x \leftarrow a) = \mathit{true}; \\ \mathit{false}, & \text{if for all } a \in \mathcal{M}(s)\text{: } [\![\phi]\!]_{\mathcal{M}}e(x \leftarrow a) = \mathit{false}; \\ \bot & \text{otherwise.} \end{cases}$$

Table 1: Three-valued Semantics

value in the condition of an if-then-else provides the user with useful information, namely, that the program may crash during execution under certain conditions. For example, consider the following piece of C code:

```
int *p = malloc(sizeof(int));
int x = (*p > 0)? y : z;
```

In this example, the if-then-else operator (which is `(·)?  ·:·` in C) will cause the program to crash if `p` happens to be `NULL`, even if $y = z$ in this particular program state. Here `*p` is a partial function defined over non-null pointers to integers, and returning an integer.

The logical if-then-else is defined similarly to the term if-then-else, so that DeMorgan law for negation and the if-lifting properties for any predicate symbol p in Σ are preserved:

$$\begin{aligned}\neg(\mathbf{if}\ \phi\ \mathbf{then}\ \phi_1\ \mathbf{else}\ \phi_2\ \mathbf{endif}) &\approxeq \mathbf{if}\ \phi\ \mathbf{then}\ \neg\phi_1\ \mathbf{else}\ \neg\phi_2\ \mathbf{endif}\\ p(\mathbf{if}\ \phi\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\ \mathbf{endif}) &\approxeq \mathbf{if}\ \phi\ \mathbf{then}\ p(t_1)\ \mathbf{else}\ p(t_2)\ \mathbf{endif}\end{aligned}$$

7.3 Three-Valued Validity

The three-valued semantics can be extended to validity of formulas in the following way. A formula is considered *valid*, if in all models $\mathcal{M}$ and for all variable assignments e, $[\![\phi]\!]_{\mathcal{M}}e = \mathit{true}$. A formula is *invalid* if there is at least one such model $\mathcal{M}$ and one such assignment e that $[\![\phi]\!]_{\mathcal{M}}e = \mathit{false}$. Otherwise (if the formula always evaluates to either *true* or $\perp$) the validity is undefined. We denote the three-valued validity as $\models \phi$, which may hold, not hold, or be undefined.

8 Reduction from Three-Valued to Two-Valued Logic

Suppose we wish to determine the three-valued validity of some Σ-formula ϕ. Our general strategy is first to compute a formula called a Type Correctness Condition (TCC) which can be used to check whether ϕ can ever be undefined. If this check succeeds, that is, ϕ is always defined, we can then check the original formula. Both of these checks can be done using standard two-valued logic. To justify this claim, we first introduce TCCs and then show how they can be used to determine three-valued validity.

8.1 Type correctness conditions (TCCs).

A *Type Correctness Condition* for a formula ϕ of our three-valued logic is a formula which evaluates to true iff ϕ is not undefined.

First, observe that if we have a term $f(x)$, then by definition its TCC is simply $\delta_f[x]$. We can generalize this to arbitrary terms or formulas quite easily. Table 2 gives a recursive definition of $\mathcal{D}_\phi$, the TCC for an arbitrary formula ϕ.

The TCC not only identifies whether or not the formula ϕ is defined, but it can also be used to reduce the three-valued evaluation of ϕ to an evaluation in standard two-valued logic with total models.

Suppose $\mathcal{M}$ is a model of Σ. Let $\widehat{\Sigma}$ be equivalent to Σ except that all of its domain formulas are true (we call such a signature a *total* signature and a corresponding model a *total* model). Let $\widehat{\mathcal{M}}$ be a (total) model of $\widehat{\Sigma}$ whose interpretation of function and predicate symbols agrees with $\mathcal{M}$ wherever the domain formulas of $\mathcal{M}$ are true (we call $\widehat{\mathcal{M}}$ an *extension* of $\mathcal{M}$). Finally, let $[\![S]\!]^2_{\widehat{\mathcal{M}}}e$ denote the evaluation of an expression S in the model $\widehat{\mathcal{M}}$ using standard two-valued semantics. The following

$$
\begin{array}{rcl}
\mathcal{D}_x & \equiv & \mathsf{true} \\
\mathcal{D}_c & \equiv & \mathsf{true} \\
\mathcal{D}_{f(t_1,\ldots,t_n)} & \equiv & \delta_f[t_1,\ldots,t_n] \wedge \bigwedge_{i=1}^{n} \mathcal{D}_{t_i} \\
\mathcal{D}_{\mathbf{if}\ \phi\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\ \mathbf{endif}} & \equiv & \mathcal{D}_\phi \wedge (\mathbf{if}\ \phi\ \mathbf{then}\ \mathcal{D}_{t_1}\ \mathbf{else}\ \mathcal{D}_{t_2}\ \mathbf{endif}) \\
\mathcal{D}_{\mathbf{if}\ \phi\ \mathbf{then}\ \phi_1\ \mathbf{else}\ \phi_2\ \mathbf{endif}} & \equiv & \mathcal{D}_\phi \wedge (\mathbf{if}\ \phi\ \mathbf{then}\ \mathcal{D}_{\phi_1}\ \mathbf{else}\ \mathcal{D}_{\phi_2}\ \mathbf{endif}) \\
\mathcal{D}_{p(t_1,\ldots,t_n)} & \equiv & \delta_p[t_1,\ldots,t_n] \wedge \bigwedge_{i=1}^{n} \mathcal{D}_{t_i} \\
\mathcal{D}_{t_1 \approx t_2} & \equiv & \mathcal{D}_{t_1} \wedge \mathcal{D}_{t_2} \\
\mathcal{D}_{\neg\phi} & \equiv & \mathcal{D}_\phi \\
\mathcal{D}_{\phi_1 \wedge \phi_2} & \equiv & (\mathcal{D}_{\phi_1} \wedge \neg\phi_1) \vee (\mathcal{D}_{\phi_2} \wedge \neg\phi_2) \vee (\mathcal{D}_{\phi_1} \wedge \mathcal{D}_{\phi_2}) \\
\mathcal{D}_{\phi_1 \vee \phi_2} & \equiv & (\mathcal{D}_{\phi_1} \wedge \phi_1) \vee (\mathcal{D}_{\phi_2} \wedge \phi_2) \vee (\mathcal{D}_{\phi_1} \wedge \mathcal{D}_{\phi_2}) \\
\mathcal{D}_{\forall x.\,\phi} & \equiv & (\exists x.\,\mathcal{D}_\phi \wedge \neg\phi) \vee (\forall x.\,\mathcal{D}_\phi) \\
\mathcal{D}_{\exists x.\,\phi} & \equiv & (\exists x.\,\mathcal{D}_\phi \wedge \phi) \vee (\forall x.\,\mathcal{D}_\phi)
\end{array}
$$

Table 2: Definition of TCCs for terms and formulas.

two theorems justify our use of TCCs.

8.2 Main Theorems

In the proofs of the theorems we use the following simplifying device: It is clear by definition of the domain formulas $\delta_f[\,]$ and $\delta_p[\,]$ that for each function symbol f and predicate symbol p we can introduce a new signature symbol (which we will also call δ_f and δ_p,) such that for all terms $t_1, \ldots t_n$, $[\![\delta_f(t_1,\ldots,t_n)]\!]_{\mathcal{M}}e = [\![\delta_f[t_1,\ldots,t_n]]\!]_{\mathcal{M}}e$ and $[\![\delta_p(t_1,\ldots,t_n)]\!]_{\mathcal{M}}e = [\![\delta_p[t_1,\ldots,t_n]]\!]_{\mathcal{M}}e$. These new symbols can be given their natural interpretation in the models $\mathcal{M}$ and $\widehat{\mathcal{M}}$, and in fact, by their totality, $\mathcal{M}(\delta_f) = \widehat{\mathcal{M}}(\delta_f)$ and $\mathcal{M}(\delta_p) = \widehat{\mathcal{M}}(\delta_p)$

For clarity, we also distinguish explicitly between two- or three-valued semantics by using the superscript 2 or 3.

Theorem 8.1. *Let S be any Σ-term or formula, and let $\widehat{\mathcal{M}}$ denote an arbitrary* *extention of a Σ-model $\mathcal{M}$ to a total model over $\widehat{\Sigma}$. Then:*

$$[\![\mathcal{D}_S]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{true} \qquad \mathit{implies} \qquad [\![S]\!]^2_{\widehat{\mathcal{M}}}e = [\![S]\!]^3_{\mathcal{M}}e$$

Proof. (by structural induction on terms and formulas)

- $S = c : [\![\mathcal{D}_c]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{true}$, and $[\![c]\!]^2_{\widehat{\mathcal{M}}}e = \widehat{\mathcal{M}}(c) = \mathcal{M}(c) = [\![c]\!]^3_{\mathcal{M}}e$
- $S = x : [\![\mathcal{D}_x]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{true}$, and $[\![x]\!]^2_{\widehat{\mathcal{M}}}e = e(x) = [\![x]\!]^3_{\mathcal{M}}e$
- $S = f(t_1, \ldots, t_n) : \quad \mathcal{D}_S = \bigwedge_{i=1}^n \mathcal{D}_{t_i} \wedge \delta_f(t_1, \ldots, t_n)$

  By assumption $[\![\mathcal{D}_S]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{true}$, so: $[\![\delta_f(t_1, \ldots, t_n)]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{true}$ and $[\![\mathcal{D}_{t_i}]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{true}$, so by Induction Hypothesis $[\![t_i]\!]^2_{\widehat{\mathcal{M}}}e = [\![t_i]\!]^3_{\mathcal{M}}e$, for all $i = 1..n$.

$$\begin{aligned} [\![\delta_f(t_1, \ldots, t_n)]\!]^3_{\mathcal{M}}e &= \mathcal{M}(\delta_f)([\![t_1]\!]^3_{\mathcal{M}}e, \ldots, [\![t_n]\!]^3_{\mathcal{M}}e) \qquad (\delta_f \ \mathit{is\ total}) \\ &= \widehat{\mathcal{M}}(\delta_f)([\![t_1]\!]^2_{\widehat{\mathcal{M}}}e, \ldots, [\![t_n]\!]^2_{\widehat{\mathcal{M}}}e) \\ &= [\![\delta_f(t_1, \ldots, t_n)]\!]^2_{\widehat{\mathcal{M}}}e \\ &= \mathit{true} \quad (*) \end{aligned}$$

  (a) By definition, we have:

$$\begin{aligned} [\![f(t_1, \ldots, t_n)]\!]^3_{\mathcal{M}}e &= \text{if } [\![\delta_f(t_1, \ldots, t_n)]\!]^3_{\mathcal{M}}e = \mathit{true} \\ &\quad\ \text{then } \mathcal{M}(f)([\![t_1]\!]^3_{\mathcal{M}}e, \ldots, [\![t_n]\!]^3_{\mathcal{M}}e) \text{ else } \perp \\ &= \mathcal{M}(f)([\![t_1]\!]^2_{\widehat{\mathcal{M}}}e, \ldots, [\![t_n]\!]^2_{\widehat{\mathcal{M}}}e) \ \ (\mathit{by}\ *) \end{aligned}$$

  We also have by the standard definition of $[\![\ ]\!]^2$:

  (b) $[\![f(t_1, \ldots, t_n)]\!]^2_{\widehat{\mathcal{M}}}e = \widehat{\mathcal{M}}(f)([\![t_1]\!]^2_{\widehat{\mathcal{M}}}e, \ldots, [\![t_n]\!]^2_{\widehat{\mathcal{M}}}e)$

  Since $[\![\delta_f(t_1, \ldots, t_n)]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{true}$, i.e. f is defined on these terms in both $\mathcal{M}$ and $\widehat{\mathcal{M}}$ we have:

$$\widehat{\mathcal{M}}(f)([\![t_1]\!]^2_{\widehat{\mathcal{M}}}e,\ldots,[\![t_n]\!]^2_{\widehat{\mathcal{M}}}e) = \mathcal{M}(f)([\![t_1]\!]^2_{\widehat{\mathcal{M}}}e,\ldots,[\![t_n]\!]^2_{\widehat{\mathcal{M}}}e)$$

Therefore by (a) and (b), $[\![f(t_1,\ldots,t_n)]\!]^3_{\mathcal{M}}e = [\![f(t_1,\ldots,t_n)]\!]^2_{\widehat{\mathcal{M}}}e$

- $S = \textbf{if } \phi \textbf{ then } t_1 \textbf{ else } t_2 \textbf{ endif}$: $\mathcal{D}_S = \mathcal{D}_\phi \wedge \textbf{if } \phi \textbf{ then } \mathcal{D}_{t_1} \textbf{ else } \mathcal{D}_{t_2} \textbf{ endif}$

  By assumption $[\![\mathcal{D}_S]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{true}$, so:

  (a) $[\![\textbf{if } \phi \textbf{ then } \mathcal{D}_{t_1} \textbf{ else } \mathcal{D}_{t_2} \textbf{ endif}]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{true}$

  (b) $[\![\mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{true}$, so by Induction Hypothesis, $[\![\phi]\!]^2_{\widehat{\mathcal{M}}}e = [\![\phi]\!]^3_{\mathcal{M}}e$ (2)

  From (a): if $[\![\phi]\!]^2_{\widehat{\mathcal{M}}}e$ then $[\![\mathcal{D}_{t_1}]\!]^2_{\widehat{\mathcal{M}}}e$ else $[\![\mathcal{D}_{t_2}]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{true}$.

  Case I: $[\![\phi]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{true}$. Then $[\![\mathcal{D}_{t_1}]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{true}$, and by Induction Hypothesis:

$$[\![t_1]\!]^2_{\widehat{\mathcal{M}}}e \;=\; [\![t_1]\!]^3_{\mathcal{M}}e \;\; (3)$$

  Then,

$$\begin{aligned}
[\![\textbf{if } \phi \textbf{ then } t_1 \textbf{ else } t_2 \textbf{ endif}]\!]^2_{\widehat{\mathcal{M}}}e &= \text{if } [\![\phi]\!]^2_{\widehat{\mathcal{M}}}e \text{ then } [\![t_1]\!]^2_{\widehat{\mathcal{M}}}e \text{ else } [\![t_2]\!]^2_{\widehat{\mathcal{M}}}e \\
&= [\![t_1]\!]^2_{\widehat{\mathcal{M}}}e \quad \text{(by assumption of Case I)} \\
&= [\![t_1]\!]^3_{\mathcal{M}}e \quad \text{(by (3))}
\end{aligned}$$

  On the other hand,

$$\begin{aligned}
[\![\textbf{if } \phi \textbf{ then } t_1 \textbf{ else } t_2 \textbf{ endif}]\!]^3_{\mathcal{M}}e &= \begin{cases} \bot, & \text{if } [\![\phi]\!]^3_{\mathcal{M}}e = \bot \\ [\![t_1]\!]^3_{\mathcal{M}}e, & \text{if } [\![\phi]\!]^3_{\mathcal{M}}e = \mathit{true} \\ [\![t_2]\!]^3_{\mathcal{M}}e, & \text{if } [\![\phi]\!]^3_{\mathcal{M}}e = \mathit{false} \end{cases} \\
&= [\![t_1]\!]^3_{\mathcal{M}}e \quad \text{(by (2) and by Case I)}
\end{aligned}$$

  Case II: $[\![\phi]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{false}$ is symmetric.

- $S = \textbf{if } \phi \textbf{ then } \phi_1 \textbf{ else } \phi_2 \textbf{ endif}$:   Analogous to $S = \textbf{if } \phi \textbf{ then } t_1 \textbf{ else } t_2 \textbf{ endif}$.

- $S = p(t_1, \ldots, t_n)$: Analogous to $S = f(t_1, \ldots, t_n)$.

- $S = \neg\phi$: $[\![\mathcal{D}_{\neg\phi}]\!]^2_{\widehat{\mathcal{M}}} e = [\![\mathcal{D}_{\phi}]\!]^2_{\widehat{\mathcal{M}}} e$

  By assumption, $[\![\mathcal{D}_{\phi}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$, so by the Induction Hypothesis, $[\![\phi]\!]^2_{\widehat{\mathcal{M}}} e = [\![\phi]\!]^3_{\mathcal{M}} e$, and thus by definition of $[\![\ ]\!]^3$:

$$[\![\neg\phi]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{not}\ [\![\phi]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{not}\ [\![\phi]\!]^3_{\mathcal{M}} e = [\![\neg\phi]\!]^3_{\mathcal{M}} e$$

- $S = \phi_1 \wedge \phi_2$: $\mathcal{D}_{\phi_1 \wedge \phi_2} = (\mathcal{D}_{\phi_1} \wedge \neg\phi_1) \vee (\mathcal{D}_{\phi_2} \wedge \neg\phi_2) \vee (\mathcal{D}_{\phi_1} \wedge \mathcal{D}_{\phi_2})$.

  By assumption, either:

  - Case 1: $[\![\mathcal{D}_{\phi_1}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$ and $[\![\phi_1]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$
  - Case 2: $[\![\mathcal{D}_{\phi_2}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$ and $[\![\phi_2]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$
  - Case 3: $[\![\mathcal{D}_{\phi_1}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$ and $[\![\mathcal{D}_{\phi_2}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$

  Case 1: by Induction Hypothesis, $[\![\phi_1]\!]^2_{\widehat{\mathcal{M}}} e = [\![\phi_1]\!]^3_{\mathcal{M}} e$, therefore $[\![\phi_1]\!]^3_{\mathcal{M}} e = \mathit{false}$. Thus, $[\![\phi_1 \wedge \phi_2]\!]^3_{\mathcal{M}} e = \mathit{false}$ and $[\![\phi_1 \wedge \phi_2]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$, so they are equal.

  Case 2 is symmetric.

  Case 3: by Induction Hypothesis, $[\![\phi_1]\!]^2_{\widehat{\mathcal{M}}} e = [\![\phi_1]\!]^3_{\mathcal{M}} e$ and $[\![\phi_2]\!]^2_{\widehat{\mathcal{M}}} e = [\![\phi_2]\!]^3_{\mathcal{M}} e$, so both are not $\perp$. Therefore, by definition of $[\![\ ]\!]^2$ and $[\![\ ]\!]^3$: $[\![\phi_1 \wedge \phi_2]\!]^2_{\widehat{\mathcal{M}}} e = [\![\phi_1 \wedge \phi_2]\!]^3_{\mathcal{M}} e$

- $S = \phi_1 \vee \phi_2$: $\mathcal{D}_{\phi_1 \vee \phi_2} = (\mathcal{D}_{\phi_1} \wedge \phi_1) \vee (\mathcal{D}_{\phi_2} \wedge \phi_2) \vee (\mathcal{D}_{\phi_1} \wedge \mathcal{D}_{\phi_2})$.

  By assumption, either:

  - Case 1: $[\![\mathcal{D}_{\phi_1}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$ and $[\![\phi_1]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$
  - Case 2: $[\![\mathcal{D}_{\phi_2}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$ and $[\![\phi_2]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$
  - Case 3: $[\![\mathcal{D}_{\phi_1}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$ and $[\![\mathcal{D}_{\phi_2}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$

Case 1: by Induction Hypothesis, $[\![\phi_1]\!]^2_{\widehat{\mathcal{M}}}e = [\![\phi_1]\!]^3_{\mathcal{M}}e$, therefore $[\![\phi_1]\!]^3_{\mathcal{M}}e = \mathit{true}$. Thus, $[\![\phi_1 \vee \phi_2]\!]^3_{\mathcal{M}}e = \mathit{true}$ and $[\![\phi_1 \vee \phi_2]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{true}$, so they are equal.

Case 2 is symmetric.

Case 3: by Induction Hypothesis, $[\![\phi_1]\!]^2_{\widehat{\mathcal{M}}}e = [\![\phi_1]\!]^3_{\mathcal{M}}e$ and $[\![\phi_2]\!]^2_{\widehat{\mathcal{M}}}e = [\![\phi_2]\!]^3_{\mathcal{M}}e$, so both are not $\perp$. Therefore, by definition of $[\![\ ]\!]^2$ and $[\![\ ]\!]^3$: $[\![\phi_1 \vee \phi_2]\!]^2_{\widehat{\mathcal{M}}}e = [\![\phi_1 \vee \phi_2]\!]^3_{\mathcal{M}}e$

- $S = \forall x : s.\, \phi :\quad \mathcal{D}_S = (\exists x : s. \mathcal{D}_\phi \wedge \neg\phi) \vee (\forall x : s.\, \mathcal{D}_\phi)$

  By assumption, either $[\![\exists x : s.\mathcal{D}_\phi \wedge \neg\phi]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{true}$ or $[\![\forall x : s.\, \mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{true}$. In the first case, there exists $d \in \mathcal{M}(s)$ such that:

  $$[\![\mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}}e(x \leftarrow d) = \mathit{true} \text{ and } [\![\phi]\!]^2_{\widehat{\mathcal{M}}}e(x \leftarrow d) = \mathit{false}.$$

  In this case $[\![\forall x : s.\, \phi]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{false}$ and by Indiction Hypothesis, $[\![\phi]\!]^3_{\mathcal{M}}e(x \leftarrow d) = \mathit{false}$, so also $[\![\forall x : s.\, \phi]\!]^3_{\mathcal{M}}e = \mathit{false}$, i.e.: $[\![\forall x : s.\, \phi]\!]^2_{\widehat{\mathcal{M}}}e = [\![\forall x : s.\, \phi]\!]^3_{\mathcal{M}}e$.

  In the latter case, for all $d \in \mathcal{M}(s)$: $[\![\mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}}e(x \leftarrow d) = \mathit{true}$, so by the Induction Hypothesis, for all $d \in \mathcal{M}(s)$: $[\![\phi]\!]^2_{\widehat{\mathcal{M}}}e(x \leftarrow d) = [\![\phi]\!]^3_{\mathcal{M}}e(x \leftarrow d)$. Thus, by definition of $[\![\ ]\!]^3$, $[\![\forall x : s.\, \phi]\!]^2_{\widehat{\mathcal{M}}}e = [\![\forall x : s.\, \phi]\!]^3_{\mathcal{M}}e$

- $S = \exists x : s.\, \phi :\quad \mathcal{D}_S = (\exists x : s. \mathcal{D}_\phi \wedge \phi) \vee (\forall x : s.\, \mathcal{D}_\phi)$

  By assumption, either $[\![\exists x : s.\mathcal{D}_\phi \wedge \phi]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{true}$ or $[\![\forall x : s.\, \mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{true}$. In the first case, there exists $d \in \mathcal{M}(s)$ such that:

  $$[\![\mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}}e(x \leftarrow d) = \mathit{true} \text{ and } [\![\phi]\!]^2_{\widehat{\mathcal{M}}}e(x \leftarrow d) = \mathit{true}.$$

  In this case $[\![\exists x : s.\, \phi]\!]^2_{\widehat{\mathcal{M}}}e = \mathit{true}$ and by Indiction Hypothesis, $[\![\phi]\!]^3_{\mathcal{M}}e(x \leftarrow d) = \mathit{true}$, so also $[\![\exists x : s.\, \phi]\!]^3_{\mathcal{M}}e = \mathit{true}$, i.e.: $[\![\exists x : s.\, \phi]\!]^2_{\widehat{\mathcal{M}}}e = [\![\exists x : s.\, \phi]\!]^3_{\mathcal{M}}e$.

In the latter case, for all $d \in \mathcal{M}(s)$: $[\![\mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}} e(x \leftarrow d) = \mathit{true}$, so by the Induction Hypothesis, for all $d \in \mathcal{M}(s)$: $[\![\phi]\!]^2_{\widehat{\mathcal{M}}} e(x \leftarrow d) = [\![\phi]\!]^3_{\mathcal{M}} e(x \leftarrow d)$.

Thus, by definition of $[\![\ ]\!]^3$, $[\![\exists x : s.\, \phi]\!]^2_{\widehat{\mathcal{M}}} e = [\![\exists x : s.\, \phi]\!]^3_{\mathcal{M}} e$.

□

Theorem 8.2. *Let S be any Σ-term or formula, and let $\widehat{\mathcal{M}}$ denote an arbitrary extention of a Σ-model $\mathcal{M}$ to a total model over $\widehat{\Sigma}$. Then:*

$$[\![\mathcal{D}_S]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false} \qquad \mathit{implies} \qquad [\![S]\!]^3_{\mathcal{M}} e = \bot$$

Proof. (by structural induction on terms and formulas)

- $S = c$: Claim holds vacuously, since $[\![\mathcal{D}_c]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$
- $S = x$: Claim holds vacuously, since $[\![\mathcal{D}_x]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$
- $S = f(t_1, \ldots, t_n)$: $\quad \mathcal{D}_S = \bigwedge_{i=1}^n \mathcal{D}_{t_i} \wedge \delta_f(t_1, \ldots, t_n)$

  By assumption $[\![\mathcal{D}_S]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$, so:

  - *either*: $[\![\mathcal{D}_{t_k}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$ for some $k \in \{1, \ldots, n\}$. In this case by Induction Hypothesis $[\![t_k]\!]^3_{\mathcal{M}} e = \mathit{false}$ and by definition of $[\![\ ]\!]^3$: $[\![f(t_1, \ldots, t_n)]\!]^3_{\mathcal{M}} e = \bot$.
  - *or*: $[\![\delta_f(t_1, \ldots, t_n)]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$.

    In this case also by definition of $[\![\ ]\!]^3$: $[\![f(t_1, \ldots, t_n)]\!]^3_{\mathcal{M}} e = \bot$.

- $S = \textbf{if } \phi \textbf{ then } t_1 \textbf{ else } t_2 \textbf{ endif}$: $\quad \mathcal{D}_S = \mathcal{D}_\phi \wedge \textbf{if } \phi \textbf{ then } \mathcal{D}_{t_1} \textbf{ else } \mathcal{D}_{t_2} \textbf{ endif}$

  By assumption $[\![\mathcal{D}_S]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$, so:

  - *either*: $[\![\mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$. In this case by Induction Hypothesis $[\![\phi]\!]^3_{\mathcal{M}} e = \bot$ and by definition of $[\![\ ]\!]^3$: $[\![\textbf{if } \phi \textbf{ then } t_1 \textbf{ else } t_2 \textbf{ endif}]\!]^3_{\mathcal{M}} e = \bot$.

- *or*: $[\![\mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$ and $[\![\textbf{if } \phi \textbf{ then } \mathcal{D}_{t_1} \textbf{ else } \mathcal{D}_{t_2} \textbf{ endif}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$

  Case 1: $[\![\phi]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$. Then $[\![\mathcal{D}_{t_1}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$. By Induction Hypothesis:

  $$[\![t_1]\!]^3_{\mathcal{M}} e = \bot. \qquad (1)$$

  Also, since $[\![\phi]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$ and $[\![\mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$, by Theorem 8.1

  $$[\![\phi]\!]^3_{\mathcal{M}} e = \mathit{true}. \qquad (2)$$

  Therefore:

  $$\begin{aligned} [\![\textbf{if } \phi \textbf{ then } t_1 \textbf{ else } t_2 \textbf{ endif}]\!]^3_{\mathcal{M}} e &= \begin{cases} \bot, & \text{if } [\![\phi]\!]^3_{\mathcal{M}} e = \bot \\ [\![t_1]\!]^3_{\mathcal{M}} e, & \text{if } [\![\phi]\!]^3_{\mathcal{M}} e = \mathit{true} \\ [\![t_2]\!]^3_{\mathcal{M}} e, & \text{if } [\![\phi]\!]^3_{\mathcal{M}} e = \mathit{false} \end{cases} \\ &= \bot \quad \text{(by (1) and (2))} \end{aligned}$$

  Case 2, where $[\![\phi]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$ is symmetric.

- $S = \textbf{if } \phi \textbf{ then } \phi_1 \textbf{ else } \phi_2 \textbf{ endif}$: Analogous to $S = \textbf{if } \phi \textbf{ then } t_1 \textbf{ else } t_2 \textbf{ endif}$.

- $S = P(t_1, \ldots, t_n)$: Analogous to $S = f(t_1, \ldots, t_n)$.

- $S = \neg\phi$: $[\![\mathcal{D}_{\neg\phi}]\!]^2_{\widehat{\mathcal{M}}} e = [\![\mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}} e$

  By assumption, $[\![\mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$, so by the Induction Hypothesis, $[\![\phi]\!]^3_{\mathcal{M}} e = \bot$, and thus by definition of $[\![\ ]\!]^3$, $[\![\neg\phi]\!]^3_{\mathcal{M}} e = \bot$.

- $S = \phi_1 \wedge \phi_2$: $\mathcal{D}_{\phi_1 \wedge \phi_2} = (\mathcal{D}_{\phi_1} \wedge \neg\phi_1) \vee (\mathcal{D}_{\phi_2} \wedge \neg\phi_2) \vee (\mathcal{D}_{\phi_1} \wedge \mathcal{D}_{\phi_2})$.

  By assumption:

  - (a) $[\![\mathcal{D}_{\phi_1}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$ or $[\![\phi_1]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$
  - (b) $[\![\mathcal{D}_{\phi_2}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$ or $[\![\phi_2]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$
  - (c) $[\![\mathcal{D}_{\phi_1}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$ or $[\![\mathcal{D}_{\phi_2}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$

From (c), without loss of generality, say $[\![\mathcal{D}_{\phi_1}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$ (other case is symmetric.) Then by Induction Hypothesis:

$$[\![\phi_1]\!]^3_{\mathcal{M}} e = \bot \qquad (3)$$

From (b) we have:

- *either*: $[\![\mathcal{D}_{\phi_2}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$. By Induction Hypothesis, $[\![\phi_2]\!]^3_{\mathcal{M}} e = \bot$. Therefore, by (3): $[\![\phi_1 \wedge \phi_2]\!]^3_{\mathcal{M}} e = \bot$.
- *or*: $[\![\mathcal{D}_{\phi_2}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$, and $[\![\phi_2]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$. By Theorem 8.1, $[\![\phi_2]\!]^3_{\mathcal{M}} e = \mathit{true}$, so in this case also using (3): $[\![\phi_1 \wedge \phi_2]\!]^3_{\mathcal{M}} e = \bot$.

- $S = \phi_1 \vee \phi_2$: $\quad \mathcal{D}_{\phi_1 \vee \phi_2} = (\mathcal{D}_{\phi_1} \wedge \phi_1) \vee (\mathcal{D}_{\phi_2} \wedge \phi_2) \vee (\mathcal{D}_{\phi_1} \wedge \mathcal{D}_{\phi_2})$.

By assumption:

- (a) $[\![\mathcal{D}_{\phi_1}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$ or $[\![\phi_1]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$
- (b) $[\![\mathcal{D}_{\phi_2}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$ or $[\![\phi_2]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$
- (c) $[\![\mathcal{D}_{\phi_1}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$ or $[\![\mathcal{D}_{\phi_2}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$

From (c), without loss of generality, say $[\![\mathcal{D}_{\phi_1}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$ (other case is symmetric.) Then by Induction Hypothesis:

$$[\![\phi_1]\!]^3_{\mathcal{M}} e = \bot \qquad (4)$$

From (b) we have:

- *either*: $[\![\mathcal{D}_{\phi_2}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$. By Induction Hypothesis, $[\![\phi_2]\!]^3_{\mathcal{M}} e = \bot$, Therefore, using (4): $[\![\phi_1 \vee \phi_2]\!]^3_{\mathcal{M}} e = \bot$.
- *or*: $[\![\mathcal{D}_{\phi_2}]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$, and $[\![\phi_2]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$. By Theorem 8.1, $[\![\phi_2]\!]^3_{\mathcal{M}} e = \mathit{false}$, so in this case also using (4): $[\![\phi_1 \vee \phi_2]\!]^3_{\mathcal{M}} e = \bot$.

- $S = \forall x : s.\, \phi :\quad \mathcal{D}_S = (\exists x : s. \mathcal{D}_\phi \wedge \neg\phi) \vee (\forall x : s.\, \mathcal{D}_\phi)$ By assumption:

  (a) $[\![\exists x : s.\mathcal{D}_\phi \wedge \neg\phi]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$

  (b) $[\![\forall x : s.\, \mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$.

  From (b), there exists $d_0 \in \mathcal{M}(s)$ such that: $[\![\mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}} e(x \leftarrow d_0) = \mathit{false}$, so by Induction Hypothesis:

  $$[\![\phi]\!]^3_{\mathcal{M}} e(x \leftarrow d_0) = \bot \qquad (5)$$

  From (a) for all $d \in \mathcal{M}(s)$:

  - *either*: $[\![\mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}} e(x \leftarrow d) = \mathit{false}$.

    In this case, by Induction Hypothesis, $[\![\phi]\!]^3_{\mathcal{M}} e(x \leftarrow d) = \bot$.

  - *or*: $[\![\mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}} e(x \leftarrow d) = \mathit{true}$, and $[\![\phi]\!]^2_{\widehat{\mathcal{M}}} e(x \leftarrow d) = \mathit{true}$.

    Therefore, by Theorem 8.1 $[\![\phi]\!]^3_{\mathcal{M}} e(x \leftarrow d) = \mathit{true}$.

  In both cases, using (5) and definition of $[\![\ ]\!]^3$ we obtain: $[\![\forall x : s.\, \phi]\!]^3_{\mathcal{M}} e = \bot$.

- $S = \exists x : s.\, \phi :\quad \mathcal{D}_S = (\exists x : s. \mathcal{D}_\phi \wedge \phi) \vee (\forall x : s.\, \mathcal{D}_\phi)$

  By assumption:

  (a) $[\![\exists x : s.\mathcal{D}_\phi \wedge \phi]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$

  (b) $[\![\forall x : s.\, \mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{false}$.

  From (b), there exists $d_0 \in \mathcal{M}(s)$ such that: $[\![\mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}} e(x \leftarrow d_0) = \mathit{false}$, so by Induction Hypothesis:

  $$[\![\phi]\!]^3_{\mathcal{M}} e(x \leftarrow d_0) = \bot \qquad (6)$$

  From (a) for all $d \in \mathcal{M}(s)$:

  - *either*: $[\![\mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}} e(x \leftarrow d) = \mathit{false}$.

    In this case, by Induction Hypothesis, $[\![\phi]\!]^3_{\mathcal{M}} e(x \leftarrow d) = \bot$.

- *or*: $[\![\mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}} e(x \leftarrow d) = \mathit{true}$, and $[\![\phi]\!]^2_{\widehat{\mathcal{M}}} e(x \leftarrow d) = \mathit{false}$.

  Therefore, by Theorem 8.1 $[\![\phi]\!]^3_{\mathcal{M}} e(x \leftarrow d) = \mathit{false}$.

In both cases, using (6) and definition of $[\![\ ]\!]^3$ we obtain: $[\![\exists x : s.\, \phi]\!]^3_{\mathcal{M}} e = \bot$.

$\square$

Another important property of $\mathcal{D}_\phi$ is that if ϕ is represented as a DAG, then the worst-case size of $\mathcal{D}_\phi$ as a DAG is linear in the size of ϕ. This point is discussed in our publication [10].

8.3 Checking validity

Theorems 8.1 and 8.2 and the procedure for constructing $\mathcal{D}_\phi$ effectively provide an algorithm for checking whether a formula is valid (true for all variable assignments) in a (partial) model $\mathcal{M}$. All we have to do is construct a decision procedure DP that can determine whether the formula is valid in $\widehat{\mathcal{M}}$, an arbitrary extension of $\mathcal{M}$.

Suppose we want to determine whether ϕ is true in $\mathcal{M}$. We first check $\mathcal{D}_\phi$, the TCC of ϕ. If $\mathsf{DP}(\mathcal{D}_\phi)$ is false, then $[\![\mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}} e = \mathsf{false}$ for some assignment e, so $[\![\phi]\!]_{\mathcal{M}} e = \bot$ by Theorem 8.2. Thus, ϕ is not valid in $\mathcal{M}$. On the other hand, if $\mathsf{DP}(\mathcal{D}_\phi)$ is true, then $[\![\mathcal{D}_\phi]\!]^2_{\widehat{\mathcal{M}}} e = \mathit{true}$ for all e, so $[\![\phi]\!]^2_{\widehat{\mathcal{M}}} e = [\![\phi]\!]_{\mathcal{M}} e$ for all e by Theorem 8.1. Thus, $\mathsf{DP}(\phi)$ effectively determines the validity of ϕ in $\mathcal{M}$.

This property is extremely useful from a practical implementation point of view, as we can build a decision procedure for any convenient extension of $\mathcal{M}$ in which all functions are total. Since evaluation and simplification are common steps in decision procedures, this eliminates the need to handle partial functions as special cases, and we can just evaluate or simplify them as we would any other function.

As a specific example, consider the model of arithmetic with division, where division by zero is undefined. Decision procedures for arithmetic often require being able to put terms in a normal form. In particular, it is desirable to be able to evaluate constant expressions to obtain constants. In the standard model where division is a partial function, there is no correct way to evaluate 1/0, but if we extend that model, say by defining division by 0 to be 0, then all constant expressions can easily be evaluated. Our approach shows that a decision procedure with this additional assumption can be used to decide validity in the model where division is a partial function.

Part III

Deciding Theories of Algebraic Data Types

9 Introduction and Related Work

The historically foundational decidability and quantifier elimination results for term algebras can be found in [31]. In other early work, [26] addresses the problem of satisfiability of one equation in a term algebra, modulo other equations. The applications and extension of the quantifier elimination procedure to term algebras with queues is handled in [41]. Another contribution to solving satisfiability of equations over term algebras is given in [47], which extends the language with a powerful *sub-term relation* predicate. In [22] two dual axiomatizations of term algebras are presented, one with constructors only, the other with selectors and testers only.

An often-cited reference for the quantifier-free case is the treatment by Oppen in 1980 [37]. Oppen's algorithm gives a detailed decision procedure for a single data type with a single constructor. The algorithm is linear for conjunctions of literals and NP-complete for arbitrary quantifier-free formulas. The case of multiple constructors is not addressed. In [36], Nelson and Oppen show that for a simple list data type with two constructors, satisfiability of conjunctions of literals is NP-complete. Shostak gives an algorithm for a simple theory of lists without *null* in [43].

More recently, several papers [27, 50, 51] explore decision procedures for a single algebraic data type. These papers focus on ambitious schemes for quantifier elimination and combinations with other theories rather than the question of a simple and efficient algorithm for the quantifier-free case. One possible extension of Oppen's algorithm to the case of multiple constructors is discussed briefly in [50]. A comparison of our algorithm with that of [50] is made in Section 12.

Finally, a recent approach based on first-order reasoning with the superposition calculus is described in [11]. This work shows how a decision procedure for an inductive data type with a single constructor can be automatically inferred from the first-order axioms, even though the axiomatization is infinite. While the algorithm as given is worst-case exponential, it has the advantage of being easily implementable (any existing superposition-based theorem prover can be used to implement the strategy). However, as far as the decision procedure is concerned, our focus is on generality and efficiency rather than immediacy of implementation. In our publications [6, 7] we also examine how to combine algebraic types with other arbtrary non-algebraic sorts within this decision procedure.

9.1 Type Correctness Conditions

For reasons explained in Section 4.2, we assume that associated with every selector $S_C^{(i)} : \tau \to s$ is a distinguished ground term t_C^i of sort s containing no selectors (or testers). The necessity of having these distinguished designated ground terms is closely linked with the theory of Part II. Intuitively, a selector $S_C^{(i)} : \tau \to s$ is interpreted as a partial function, since its return value is undefined when applied to a term $t = C'(\ldots)$, even if t is of type τ. For that reason, in the model $\mathcal{R}$, its return value is forced to be equal to t_C^i of type s, so that we can proceed as though the intended model is total. However, as indicated in Section 8, to reduce the validity checking to the scenario without undefined values, we need to employ the technique of TCCs. Table 3 shows what TCCs are associated with specific terms or formulas in the theory of ADTs.

$$\boxed{\begin{array}{rcl} \mathcal{D}(x) & \equiv & \textsf{true} \\ \mathcal{D}(t_1 \approx t_2) & \equiv & \mathcal{D}(t_1) \wedge \mathcal{D}(t_2) \\ \mathcal{D}(is_C(t)) & \equiv & \mathcal{D}(t) \\ \mathcal{D}(C(t_1, \ldots, t_n)) & \equiv & \bigwedge_{i=1}^{n} \mathcal{D}(t_i) \\ \mathcal{D}(S_C^{(i)}(t)) & \equiv & \mathcal{D}(t) \wedge is_C(t) \end{array}}$$

Table 3: Type Correctness Conditions for the Theory of ADTs

9.2 Contributions of this Work

There are three main contributions of this work over earlier work on the topic. First, our setting is more general: we allow mutually recursive algebraic types, each with multiple constructors, selectors, and testers, and we use the more general setting of many-sorted logic. The rationale for a many-sorted approach is that it more closely corresponds to potential applications such as analysis of programming languages. In particular, the well-sortedness requirements rule out many syntactical constructs that would not make sense in practice.

The second contribution is in presentation. We present the theory itself in terms of an initial model rather than axiomatically as is often done. Also, the presentation of the decision procedure is given as abstract rewrite rules, making it more flexible and easier to analyze than if it were given imperatively.

Finally, as described in Section 12, the flexibility provided by the abstract algorithm allows us to describe a new strategy with significantly improved practical efficiency.

Our procedure requires one additional constraint on the set of ADTs: It must be *well-founded*. A sort s is well-founded iff there exist ground (i.e., variable-free) Σ-terms of sort s. Informally, each sort must contain terms that do not denote cyclic or otherwise infinite data types. In some cases, it will be necessary to distinguish between *finite* and *infinite* sorts and constructors:

- A sort s is *finite* iff there are only finitely many ground Σ-terms of sort s;
- a constructor C is *finite* if it is nullary or if all of its argument sorts are finite.

As we will see, consistent with the above terminology, our semantics will interpret finite, resp. infinite, τ-sorts indeed as finite, resp. infinite, sets.

Subsequent sections build on the background that has been presented in Sections 4.1, 4.2, 5.1, 5.2. Recall that we denote by $\mathcal{T}(\Sigma)$ the set of (well-sorted) ground terms of signature Σ or, equivalently, the many-sorted term algebra over that signature. Also, let $\mathcal{R}$ be defined as in subsection 5.2. In Section 10, we present our decision procedure as a set of abstract rules. The correctness of the algorithm is shown in Section 11. In Section 12, we discuss the efficiency of the algorithm and show, in particular, that it can be exponentially more efficient than previous naive algorithms.

Acknowledgement. Part III has been developed based on our joint work with Clark Barrett and Cesare Tinelli.

10 The Decision Procedure

In this section, we present a decision procedure for the satisfiability of sets of Σ-literals over $\mathcal{R}$. Before giving a formal description of the algorithm, which is quite technical, we start with an informal overview based on examples.

10.1 Overview and Examples

Our procedure builds on the algorithm by Oppen [37] for a single type with a single constructor. Let us first look at how Oppen's procedure works on a simple example.

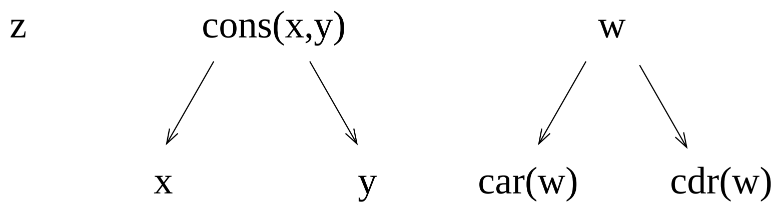

Figure 6: Term graph for Example 10.1

Example 10.1. *Consider the list data type without the null constructor*[2] *and the following set of literals:* $\{cons(x, y) \approx z, car(w) \approx x, cdr(w) \approx y, w \not\approx z\}$.

Oppen's procedure works as follows: first, a graph is constructed that relates terms according to their meaning in the intended model. The graph for Example 10.1 is shown in Figure 6. Notice that $cons(x, y)$ is a parent of x and y and $car(w)$ and $cdr(w)$ are children of w. The Oppen algorithm next computes the equivalence relation on nodes of the graph induced by the set of all equations. It then proceeds by performing an *upwards* (congruence) and *downwards* (unification) closure on the graph and then checking for cycles or for a violation of disequalities. A cycle occurs if there exists a sequence of nodes beginning and ending with the same node such that adjacent nodes are either distinct nodes in the same equivalence class or are adjacent in the graph.[3] For Example 10.1, upwards closure implies that $w \approx cons(x, y)$. But since we also have $cons(x, y) \approx z$, this contradicts the disequality $w \not\approx z$, indicating that the set of literals is unsatisfiable.

An alternative algorithm for the case of a single constructor is to introduce new terms and variables to replace variables that are inside of selectors. For Example 10.1, we would introduce $w \approx cons(s, t)$ where s, t are new variables. Now, by substituting and collapsing applications of selectors to constructors, we get

[2]Note that this data type is not well-founded. Indeed, because Oppen only considers data types with a single constructor, there is no base case for terms (unless the constructor has arity 0), so his semantics are over models with infinite terms. In contrast, we choose to disallow models with infinite terms while allowing multiple constructors, a combination that we feel is more intuitive and corresponds better to actual uses of ADTs.

[3]A simple example of a cycle is: $cons(x, y) \approx y$.

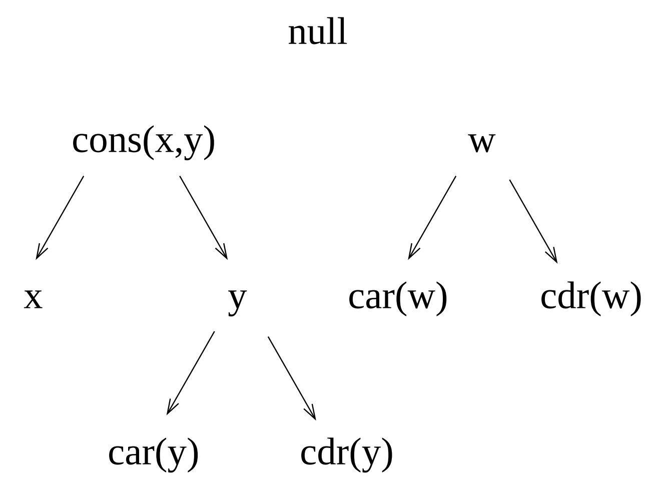

Figure 7: Term graph for Example 10.2

$\{cons(x, y) \approx z, w \approx cons(s, t), x \approx s, t \approx y, w \not\approx z\}$. This approach, advocated in [43], only requires downwards closure.

Unfortunately, if a data type has more than one constructor, things are not quite as simple. In particular, the simple approach of replacing variables with constructor terms does not work because one cannot establish *a priori* which constructor should be used to build the value denoted by a given variable.

Example 10.2. *Consider again the list data type, this time with both the cons and the null constructor, and the following set of literals:* $\{cons(x, y) \approx w, cdr(w) \approx cdr(y), y \not\approx null\}$.

The graph for Example 10.2 is shown in Figure 7. Observe that the new graph has nodes for both children of w and y, even though these terms do not all appear in the given set of literals. For the sake of simplicity, we follow Oppen in requiring that every node with at least one child has a complete set of children.

A simple extension of Oppen's algorithm for the case of multiple constructors is proposed in [50]. The idea is to first guess a *type completion*, that is, a labeling of every variable by a constructor, which is meant to constrain a variable to take only values built with the associated constructor. Once all variables are labeled by a single constructor, the Oppen algorithm can be used to determine if the constraints can be satisfied under that labeling.

Unfortunately, the type completion guess can be very expensive in practice. In Example 10.2, there are 7 terms that are not constructor terms and thus could potentially have been constructed using either constructor. A naive type completion guess would require 2^7 cases. However, most of these cases need not be considered. In fact, we only need to consider which constructor is used to construct the value of y. If y is constructed with *null*, then this contradicts the disequality $y \not\approx \mathit{null}$. On the other hand, if y is constructed with *cons*, then downward closure requires $y \approx \mathit{cdr}(w) \approx \mathit{cdr}(y)$, creating a cycle.

Our presentation combines ideas from previous work as well as introducing some new ones. There is a set of upward and downward closure rules to mimic Oppen's algorithm. The idea of a type completion is replaced by a set of labeling rules that can be used to refine the set of possible constructors for each term (in particular, this allows us to delay guessing as long as possible). And the notion of introducing constructors and eliminating selectors is captured by a set of selector rules. In addition to the presentation, one of our key contributions is to provide precise side-conditions for when case splitting is necessary as opposed to when it can be delayed. The results given in Section 12 show that with the right strategy, significant gains in efficiency can be obtained.

We describe our procedure formally in the following, as a set of derivation rules. We build on and adopt the style of similar rules for abstract congruence closure [2] and syntactic unification [32].

10.2 Definitions and Notation

In the following, we will consider well-sorted formulas over the signature Σ above and an infinite set X of implicitly existential variables. To distinguish these variables, which can occur in formulas given to the decision procedure described below, from other internal variables used by the decision procedure, we will sometimes call the elements of X *input* variables.

Given a set Γ of literals over Σ and variables from X, we wish to determine the satisfiability of Γ in the algebra $\mathcal{R}$.[4] That is, we wish to determine whether there exists a variable assignment α, a mapping of input variables to ground terms, such that applying α to Γ results in a set of ground literals all of which are true in $\mathcal{R}$. We will assume for simplicity, and with no loss of generality, that the only occurrences of terms of sort Prop are in atoms of the form $is_C(t) \approx \mathsf{true}$, which we will write just as $is_C(t)$.

Following [2], for each sort τ we will make use of the sets V_τ of *abstraction* variables of sort τ; abstraction variables are disjoint from input variables (variables in Γ) and function as equivalence class representatives for the terms in Γ. We assume an arbitrary, but fixed, well-founded ordering $\succ$ on the abstraction variables that is total on variables of the same sort. We denote the set of all variables (both input and abstraction) in Γ as $\mathcal{V}ar(\Gamma)$. Recall that for each sort τ the set $\mathcal{C}_\tau$ denotes the set of τ's constructors. We will write $sort(t)$ to denote the sort of the term t.

The rules make use of three additional constructs that are not in the language of Σ: $\rightarrow$, $\mapsto$, and *Inst*. The symbol $\rightarrow$ is used to represent *oriented* equations. Its left-hand side is a Σ-term t and its right-hand side is an abstraction variable v. The symbol $\mapsto$ denotes *labelings* of abstraction variables with sets of constructor symbols.

[4]In both theory and practice, the satisfiability of arbitrary quantifier-free formulas can be easily determined given a decision procedure for a set of literals. Using the fact that a universal formula $\forall \mathbf{x} \varphi(\mathbf{x})$ is true in a model exactly when $\neg\varphi(\mathbf{x})$ is unsatisfiable in the model, this also provides a decision procedure for universal formulas.

It is used to keep track of possible constructors for instantiating a τ variable. Finally, the *Inst* construct is used to track applications of the **Instantiate 2** rule given below. It is needed to ensure termination by preventing multiple applications of the rule. It is a unary predicate that is applied only to abstraction variables.

Let Σ^{C} denote the set of all constant symbols in Σ, including nullary constructors. We will denote by Λ the set of all possible literals over Σ and input variables X. Note that this does not include oriented equations ($t \rightarrow v$), labeling pairs ($v \mapsto L$), or applications of *Inst*. In contrast, we will denote by E multisets of literals of Λ, oriented equations, and labeling pairs, and applications of *Inst*. To simplify the presentation, we will consistently use the following meta-variables: c, d denote constants (elements of Σ^{C}) or input variables from X; u, v, w denote abstraction variables; t denotes a *flat term*—i.e., a term all of whose proper sub-terms are abstraction variables—or a label set, depending on the context. $\mathbf{u}, \mathbf{v}$ denote possibly empty sequences of abstraction variables; and $\mathbf{u} \rightarrow \mathbf{v}$ is shorthand for the set of oriented equations resulting from pairing corresponding elements from $\mathbf{u}$ and $\mathbf{v}$ and orienting them so that the left hand variable is greater than the right hand variable according to $\succ$. Finally, $v \bowtie t$ denotes any of $v \approx t$, $t \approx v$, $v \not\approx t$, $t \not\approx v$, or $v \mapsto t$. To streamline the notation, we will sometimes denote function application simply by juxtaposition.

Each rule consists of a premise and one or more conclusions. Each premise is made up of a multiset of literals from Λ, oriented equations, labeling pairs, and applications of *Inst*. Conclusions are either similar multisets or $\bot$, where $\bot$ represents a trivially unsatisfiable formula. As we show later, the soundness of our rule-based procedure depends on the fact that the premise E of a rule is satisfied in $\mathcal{R}$ by a valuation of $\mathcal{V}ar(E)$ iff one of the conclusions E' of the rule is satisfied in $\mathcal{R}$ by an extension of that valuation.

10.3 The derivation rules

Our decision procedure consists of the following derivation rules on multisets E.

Abstraction rules

Abstract 1 $$\frac{p[c],\ E}{c \to v,\ v \mapsto \mathcal{C}_\tau,\ p[v],\ E} \quad \text{if} \quad \begin{array}{l} p \in \Lambda,\ c : \tau, \\ v \text{ fresh from } V_\tau \end{array}$$

Abstract 2 $$\frac{p[C\,\mathbf{u}],\ E}{C\,\mathbf{u} \to v,\ p[v],\ v \mapsto \{C\},\ E} \quad \text{if} \ \ p \in \Lambda,\ C \in \mathcal{C}_\tau\, v \text{ fresh from } V_\tau$$

Abstract 3 $$\frac{p[S_C^{(k)}\,u],\ E}{\begin{array}{l} S_C^{(1)}\,u \to v_1, \ldots, S_C^{(n)}\,u \to v_n,\ p[v_k], \\ v_1 \mapsto \mathcal{C}_{s_1}, \ldots, v_n \mapsto \mathcal{C}_{s_n}, E \end{array}} \quad \text{if} \quad \begin{array}{l} p \in \Lambda, \\ C : s_1 \cdots s_n \to \tau, \\ \text{each } v_i \text{ fresh from } V_{s_i} \end{array}$$

The abstraction or *flattening* rules assign a new abstraction variable to every sub-term in the original set of literals. Each rule contains a literal of the form $p[t]$ in the premise and $p[v]$ in the conclusion. The meaning of this notation is that $p[t]$ is some literal containing the term t and $p[v]$ is the literal obtained by replacing every occurrence of t in $p[t]$ with the abstraction variable v. Abstraction variables are used as place-holders or equivalence class representatives for the sub-terms they replace. While we would not expect a practical implementation to actually introduce these variables, it greatly simplifies the presentation of the remaining rules.

The **Abstract 1** rule replaces input variables or constants. **Abstract 2** replaces constructor terms, and **Abstract 3** replaces selector terms. Notice that in each case, a labeling pair for the introduced variables is also created. This corresponds to labeling each sub-term with the set of possible constructors with which it could have been constructed. Also notice that in the **Abstract 3** rule, whenever a selector is applied, we effectively introduce all possible applications of selectors associated with the same constructor. This simplifies the later selector rules and corresponds to the step in the Oppen algorithm which ensures that in the term graph, any node with children has a complete set of children.

Literal level rules

Orient $$\frac{u \approx v,\ E}{u \to v,\ E} \quad \text{if } u \succ v$$

Remove 1 $$\frac{is_C\, v,\ E}{v \mapsto \{C\},\ E}$$

Inconsistent $$\frac{v \not\approx v,\ E}{\bot}$$

Remove 2 $$\frac{\neg is_C\, v,\ E}{v \mapsto \mathcal{C}_{sort(v)} \setminus \{C\},\ E}$$

The simple literal level rules are mostly self-explanatory. The **Orient** rule is used to replace an equation between abstraction variables (which every equation eventually becomes after applying the abstraction rules) with an oriented equation. Oriented equations are used in the remaining rules below. The **Inconsistent** rule detects violations of the reflexivity of equality. The **Remove** rules remove applications of testers and replace them with labeling pairs that impose the same constraints.

Upward (i.e., congruence) closure rules

Simplify 1 $$\frac{u \bowtie t,\ u \to v,\ E}{v \bowtie t,\ u \to v,\ E}$$

Simplify 2 $$\frac{f\mathbf{u}u\mathbf{v} \to w,\ u \to v,\ E}{f\mathbf{u}v\mathbf{v} \to w,\ u \to v,\ E}$$

Superpose $$\frac{t \to u,\ t \to v,\ E}{u \to v,\ t \to v,\ E} \quad \text{if } u \succ v$$

Compose $\dfrac{t \to v,\ v \to w,\ E}{t \to w,\ v \to w,\ E}$

These rules are modeled after similar rules for abstract congruence closure in [2]. The **Simplify** and **Compose** rules essentially provide a way to replace any abstraction variable with a smaller (according to $\succ$) one if the two are constrained to be equal. Note that the symbol f in the **Simplify 2** rule refers to an arbitrary function symbol from Σ. The **Superpose** rule merges two equivalence classes if they contain the same term. Congruence closure is achieved by these rules because if two terms are congruent, then after repeated applications of the first set of rules, they will become syntactically identical. Then the **Superpose** rule will merge their two equivalence classes.

Downward (i.e., unification) closure rules

Decompose $\dfrac{C\,\mathbf{u} \to v,\ C\,\mathbf{v} \to v,\ E}{C\,\mathbf{u} \to v,\ \mathbf{u} \to \mathbf{v},\ E}$

Cycle $\dfrac{C_n\,\mathbf{u_n} u \mathbf{v_n} \to u_{n-1},\ \ldots,\ C_2\,\mathbf{u_2} u_2 \mathbf{v_2} \to u_1,\ C_1\,\mathbf{u_1} u_1 \mathbf{v_1} \to u,\ E}{\perp}$ if $n \geq 1$

The main downward closure rule is the **Decompose** rule: whenever two terms with the same constructor are in the same equivalence class, their arguments must be equal. Recall that $\mathbf{u} \to \mathbf{v}$ is shorthand for the set of oriented equations resulting from pairing corresponding elements from $\mathbf{u}$ and $\mathbf{v}$ and orienting them so that the left hand variable is greater than the right hand variable according to $\succ$. The **Cycle** rule detects an inconsistency when a constructor term would have to be equivalent to one of its sub-terms.

Selector rules

Instantiate 1 $$\frac{S_C^{(1)}\, u \to u_1,\ \ldots,\ S_C^{(n)}\, u \to u_n,\ u \mapsto \{C\},\ E}{C\, u_1 \cdots u_n \to u,\ u \mapsto \{C\},\ E} \quad \text{if} \quad \begin{array}{l} C : s_1 \cdots s_n \to \tau, \\ n \geq 1 \end{array}$$

Instantiate 2 $$\frac{u \mapsto \{C\},\ E}{\begin{array}{l} C\, u_1 \cdots u_n \to u,\ u \mapsto \{C\},\ \mathit{Inst}(u), \\ u_1 \mapsto \mathcal{C}_{s_1},\ \ldots,\ u_n \mapsto \mathcal{C}_{s_n}, E \end{array}} \quad \text{if} \quad \begin{array}{l} C \text{ finite constructor}, \\ C : s_1 \cdots s_n \to \tau, \\ \mathit{Inst}(u) \notin E, \\ u_i \text{ fresh from } V_{s_i} \end{array}$$

Collapse 1 $$\frac{C\, u_1 \cdots u_n \to u,\ S_C^{(i)}\, u \to v,\ E}{C\, u_1 \cdots u_n \to u,\ u_i \approx v,\ E}$$

Collapse 2 $$\frac{S_C^{(i)} u \to v,\ u \mapsto L,\ E}{t_C^i \approx v,\ u \mapsto L,\ E} \quad \text{if} \ \ C \notin L$$

Rule **Instantiate 1** is used to eliminate selectors by replacing the argument of the selectors with a new term constructed using the appropriate constructor. Only terms that have selectors applied to them can be instantiated and then only once they are uniquely labeled. Notice that all of the selectors applied to the term are eliminated at the same time. This is why the entire set of selectors is introduced in the **Abstract 3** rule.

For completeness, a term labeled with a finite constructor must be instantiated even if no selectors are applied to that term. This is accomplished by rule **Instantiate 2**. The side conditions are similar to those in **Instantiate 1**, except that this rule *only* applies to terms labeled with finite constructors. The *Inst* predicate ensures that the rule is applied at most once for each such term.

The **Collapse** rules eliminate selectors when the result of their application can be determined. In **Collapse 1**, a selector $S_C^{(i)}$ is applied to a term constructed with constructor C. In this case, the selector expression is replaced by the appropriate argument of the constructed term. In **Collapse 2**, a selector $S_C^{(i)}$ is applied to a term which must have been constructed with a constructor other than C. In this case, the designated term t_C^i for the selector replaces the selector expression.

Labeling rules

$$\textbf{Refine} \quad \frac{v \mapsto L_1,\ v \mapsto L_2,\ E}{v \mapsto L_1 \cap L_2,\ E} \qquad\qquad \textbf{Empty} \quad \frac{v \mapsto \emptyset,\ E}{\bot} \quad \text{if } \ v : \tau$$

$$\textbf{Split 1} \quad \frac{S_C^{(i)}\, u \to v,\ u \mapsto \{C\} \cup L,\ E}{S_C^{(i)}\, u \to v,\ u \mapsto \{C\},\ E \qquad S_C^{(i)}\, u \to v,\ u \mapsto L,\ E} \quad \text{if } \ L \neq \emptyset$$

$$\textbf{Split 2} \quad \frac{u \mapsto \{C\} \cup L,\ E}{u \mapsto \{C\},\ E \qquad u \mapsto L,\ E} \quad \text{if} \quad \begin{array}{l} L \neq \emptyset, \\ \{C\} \cup L \text{ all finite constructors} \end{array}$$

The **Refine** rule simply combines labeling constraints that may arise from different sources for the same abstraction variable. **Empty** enforces the constraint that every τ-term must be constructed by some constructor. The splitting rules are used to refine the set of possible constructors for a term and are the only rules that cause branching. If a term labeled with only finite constructors cannot be eliminated in some other way, **Split 2** must be applied until it is labeled with a single constructor. For other terms, the **Split 1** rule only needs to be applied to distinguish the case of a selector being applied to the "right" constructor from a selector being applied to the "wrong" constructor. On either branch, one of the **Collapse** rules will apply immediately. We discuss this further in Section 12, below.

$null \rightarrow v_1$	$v_1 \mapsto \{null\}$	$v_5 \rightarrow v_4$
$x \rightarrow v_2$	$v_2 \mapsto \{cons, null\}$	$v_9 \rightarrow v_7$
$y \rightarrow v_3$	$v_3 \mapsto \{cons, null\}$	$v_3 \not\approx v_1$
$cons(v_2, v_3) \rightarrow v_4$	$v_4 \mapsto \{cons\}$	
$w \rightarrow v_5$	$v_5 \mapsto \{cons, null\}$	
$car(v_5) \rightarrow v_6$	$v_6 \mapsto \{cons, null\}$	
$cdr(v_5) \rightarrow v_7$	$v_7 \mapsto \{cons, null\}$	
$car(v_3) \rightarrow v_8$	$v_8 \mapsto \{cons, null\}$	
$cdr(v_3) \rightarrow v_9$	$v_9 \mapsto \{cons, null\}$	

Figure 8: Example 10.2 after **Abstraction** and **Orient**

10.4 An Example Using the Rules

Let us revisit Example 10.2 and see how the rules work on this example. Recall that we have the following set of literals: $\{cons(x, y) \approx w, cdr(w) \approx cdr(y), y \not\approx null\}$. After applying the **Abstraction** and **Orient** rules, we have the set of literals shown in Figure 8. Next, the **Simplify** and **Compose** rules can be used to replace all occurrences in the first two columns of v_5 and v_9 with v_4 and v_7 respectively. Then, **Refine** can be used to eliminate two of the labeling pairs. Notice that after replacing v_5 with v_4, v_4 can be instantiated (the side conditions of **Instantiate 1** are satisfied). The resulting set of literals is shown in Figure 9. At this point, there are two *cons* terms equivalent to v_4, so the **Decompose** rule applies, yielding two new oriented equations: $v_6 \rightarrow v_2$ and $v_7 \rightarrow v_3$. These can again be used together with the congruence rules and **Refine** to simplify the other literals. The resulting set is shown in Figure 10.

At this point, the only rule that can be applied is the **Split 1** rule. And only v_3 satisfies the necessary condition of having a selector applied to it. There are two cases. Consider first the case where $v_3 \mapsto \{cons\}$. In this case, **Instantiate 1** applies, yielding $cons(v_8, v_3) \rightarrow v_3$ which yields $\bot$ by the **Cycle** rule. In the other case, we have $v_3 \mapsto \{null\}$. This time, since *null* is a finite constructor, we can apply **Instantiate 2** to get $null \rightarrow v_3$. The **Superpose** rule then gives $v_3 \rightarrow v_1$. This can

$null \to v_1$	$v_1 \mapsto \{null\}$	$v_5 \to v_4$
$x \to v_2$	$v_2 \mapsto \{cons, null\}$	$v_9 \to v_7$
$y \to v_3$	$v_3 \mapsto \{cons, null\}$	$v_3 \not\approx v_1$
$cons(v_2, v_3) \to v_4$	$v_4 \mapsto \{cons\}$	
$w \to v_4$	$v_6 \mapsto \{cons, null\}$	
$cons(v_6, v_7) \to v_4$	$v_7 \mapsto \{cons, null\}$	
$car(v_3) \to v_8$	$v_8 \mapsto \{cons, null\}$	
$cdr(v_3) \to v_7$		

Figure 9: Figure 8 after congruence rules, **Refine**, and **Instantiate 1**

$null \to v_1$	$v_1 \mapsto \{null\}$	$v_5 \to v_4$
$x \to v_2$	$v_2 \mapsto \{cons, null\}$	$v_9 \to v_3$
$y \to v_3$	$v_3 \mapsto \{cons, null\}$	$v_3 \not\approx v_1$
$cons(v_2, v_3) \to v_4$	$v_4 \mapsto \{cons\}$	
$car(v_3) \to v_8$	$v_8 \mapsto \{cons, null\}$	
$cdr(v_3) \to v_3$		
$v_6 \to v_2$		
$v_7 \to v_3$		

Figure 10: Figure 9 after **Decompose** and congruence rules

be used together with $v_3 \not\approx v_1$ to deduce $\perp$ (via the **Simplify 1** and **Inconsistent** rules).

11 Correctness

The satisfiability in $\mathcal{R}$ of a set Γ of Σ-literals with variables in X can be checked by applying exhaustively to Γ the derivation rules in the previous section. This set of rules is very flexible in that the rules can be applied in *any* order and still yield a decision procedure for the satisfiability in $\mathcal{R}$. No specific rule application strategy is needed to achieve termination, soundness or completeness. We formalize this in the following in terms of a suitable notion of *derivation* for these rules.

A *derivation tree* (for a set Γ of Σ-literals with variables in X) is a finite tree with root Γ such that for each internal node E of the tree, its children are the conclusions of some rule whose premise is E. A *refutation tree* (for Γ) is a derivation tree all of whose leaves are $\bot$. We say that a node in a derivation tree is *(ir)reducible* if (n)one of the derivation rules applies to it. A *derivation* is a sequence of derivation trees starting with the single-node tree containing Γ, where each tree is derived from the previous one by the application of a rule to one of its leaves. A *refutation* is a finite derivation ending with a refutation tree.

For a multiset E of literals, a variable assignment α is a mapping from $\mathcal{V}ar(E)$ into the elements of $\mathcal{R}$ that is well-sorted (i.e., $sort(x) = sort(\alpha(x))$ for every $x \in \mathcal{V}ar(E)$). If α is a variable assignment, then we denote by $\overline{\alpha}$ the homomorphic extension of α that maps arbitrary terms into elements of $\mathcal{R}$. We say that α satisfies $s \approx t$ iff $\overline{\alpha}(s)$ equals $\overline{\alpha}(t)$.

For convenience, we extend the notion of satisfiability and well-sortedness to the extra-logical constructs. The oriented equation $t \to v$ is well-sorted iff t and v have the same sort. Furthermore, α satisfies $t \to v$ in $\mathcal{R}$ iff α satisfies the equation $t \approx v$ in $\mathcal{R}$. The expression $v \mapsto L$, labeling a variable v of sort s with the set L of constructor symbols, is considered to be well-sorted if $L \subseteq \mathcal{C}_s$. The valuation α satisfies a labeling pair $v \mapsto L$ in $\mathcal{R}$ if α satisfies the formula $is_C(v) \approx \mathsf{true}$ for some $C \in L$. An application of *Inst* is always well-sorted and satisfied by every variable assignment. We start with a lemma that gives a couple of useful invariants.

Lemma 11.1. *Let $E_0, E_1, \ldots,$ be a branch on a derivation tree. Then the following holds for all $i \geq 0$.*

1. *If E_0 is well-sorted, then for all i, E_i is well-sorted.*

2. *For all $u \to v \in E_i$, we have $u \succ v$.*

Proof. A simple examination of each of the rules confirms that these invariants are maintained. □

Before proving termination, we need the following additional notation. For each constructor $C \in \Sigma$, let $|C|$ denote 0 if C is infinite and otherwise denote the size of the (finite) set containing all ground constructor terms whose top symbol is C, and all of their sub-terms.

Proposition 11.2 (Termination)**.** *Every derivation is finite.*

Proof. Given a derivation tree, let $E_0, E_1, \ldots$ be any branch of the tree that does not end with $\perp$. It is enough to show that the branch can be mapped to a strictly descending sequence in a well-founded ordering. The ordering $\succ_{\mathrm{l}}$ we will use is a lexicographic ordering over tuples of the form (s, t, S, T, M, A, n) where s, t, T, and n are natural numbers, S is a multiset of naturals, M is a multiset of symbols from Σ and variables from X, and A is a multiset of abstraction variables. The ordering $\succ_{\mathrm{l}}$ is the one induced by the well-founded orderings $>, >, >_{\mathrm{m}}, >, \sqsupset_{\mathrm{m}}, \succ_{\mathrm{m}}, >$ where

- $>$ is the usual ordering of the natural numbers,
- $>_{\mathrm{m}}$ is the multiset ordering induced by $>$,
- $\sqsupset_{\mathrm{m}}$ is the multiset ordering induced by some arbitrary well-founded ordering of the set $\Sigma \cup X$, and
- $\succ_{\mathrm{m}}$ is the multiset ordering induced by the given ordering $\succ$ over the abstraction variables.

The descending sequence $(s_i, t_i, S_i, T_i, M_i, A_i, n_i)$ for $i = 0, 1, \ldots$ is defined as follows. Recall that Σ-literals do not include oriented equations, labeling pairs, or applications of *Inst*.

- s_i is the number of selector symbols in the Σ-literals of E_i;
- t_i is total number of selector symbols appearing in E_i;

- S_i is the multiset consisting of the *sizes* of the Σ-literals of E_i, where by size we mean the number of occurrences of symbols from Σ (including $\approx$) and input variables, but not of abstraction variables;

- T_i is the sum of all $|v|_i$ for all abstraction variables $v \in \mathcal{V}ar(E_i)$ that do not appear as an argument to *Inst* in E_i where, for each v, $|v|_i = \sum_{C \in L_i} |C|$ and L_i is the union of all label sets for v in E_i;

- M_i is the multiset of occurrences of symbols from Σ and input variables from X in Σ-literals or oriented equations of E_i;

- A_i is the multiset of all the occurrences of abstraction variables in E_i;

- n_i is the number of label occurrences in E_i, that is, occurrences of the constructor symbols in labeling pairs of E_i.

We show that for all consecutive nodes E_i, E_{i+1} in the branch:

$$(s_i, t_i, S_i, T_i, M_i, A_i, n_i) \succ_1 (s_{i+1}, t_{i+1}, S_{i+1}, T_{i+1}, M_{i+1}, A_{i+1}, n_{i+1}).$$

The proof is by cases, depending on the rule used to derive E_{i+1} from E_i.

1. The cases corresponding to the rules **Inconsistent**, **Cycle**, and **Empty** do not apply since they all have conclusion $\perp$.

2. Suppose one of the rules **Abstract 1**, **Abstract 2**, **Orient**, **Remove 1**, or **Remove 2** was applied. Each of these rules leaves s_i and t_i unchanged while removing at least one Σ-symbol or input variable from a literal (without changing the other literals). In each of these cases, $S_i >_{\mathrm{m}} S_{i+1}$.

3. With **Abstract 3**, the number of selector symbols appearing in literals is reduced by one, so $s_i > s_{i+1}$.

4. With all the congruence closure rules except for **Superpose** when the term t in the rule is not an abstraction variable, the only change is the replacement of an abstraction variable by another abstraction variable which is smaller by Lemma 11.1(2). Thus, s_i, t_i, S_i, T_i, and M_i remain the same, while $A_i \succ_{\mathrm{m}} A_{i+1}$. In the case where **Superpose** is applied and t is not an abstraction variable, t must contain a symbol from $\Sigma \cup X$. If t contains a selector, then $s_i = s_{i+1}$ and $t_i > t_{i+1}$. Otherwise, $M_i \sqsupset_{\mathrm{m}} M_{i+1}$ (it is easy to see that s_i, t_i, S_i, and T_i remain the same in this case).

5. The **Decompose** rule does not change the values of s_i, t_i, S_i, or T_i. However, it does eliminate one occurrence of a constructor symbol. Hence, $M_i \sqsupset_{\mathrm{m}} M_{i+1}$.

6. Now consider the selector rules. With **Instantiate 1**, since the constructor C in the rule has positive arity (i.e., $n \geq 1$) then $s_i = s_{i+1}$ and $t_i > t_{i+1}$. With **Instantiate 2**, s_i, t_i and S_i are unchanged but

$$T_{i+1} = (T_i - |u|_i) + \sum_{k=1}^{n} |u_k|_{i+1} .$$

   It is not difficult to see that $|u|_i > \sum_{k=1}^{n} |u_k|_{i+1}$. Thus, $T_i > T_{i+1}$.

7. With the collapse rules, exactly one selector symbol is eliminated from (a non-literal of) E_i, so $s_i = s_{i+1}$ and $t_i > t_{i+1}$.[5]

8. Finally, consider the labeling rules. The **Refine** rule eliminates an occurrence of an abstraction variable. Hence certainly $A_i \succ_{\mathrm{m}} A_{i+1}$. All the preceding components of the tuple are unchanged with the possible exception of T_i which may get smaller when $L_1 \neq L_2$. The split rules both produce two conclusions, each of which has fewer constructors appearing in labels than in the premise.

[5]Note that $s_i = s_{i+1}$ with **Collapse 2** because, by definition, t_C^i is a ground term with no selectors.

Furthermore, this is the only change, so T_i either decreases or is unchanged, $n_i > n_{i+1}$ and everything else is unchanged.

$\square$

The soundness of the decision procedure is based on the following result.

Lemma 11.3. *The premise E of a derivation rule is satisfied in $\mathcal{R}$ by a valuation α of $\mathcal{V}ar(E)$ iff one of the conclusions E' of the rule is satisfied in $\mathcal{R}$ by an extension of α to $\mathcal{V}ar(E')$.*

Proof. Again, the proof is by cases.

(Abstraction rules) The if direction is immediate. For the other direction, for **Abstract 1**, suppose that the premise is satisfied by α in $\mathcal{R}$. We extend α by setting v to the value of c under $\mathcal{R}, \alpha$. Consider the labeling pair $v \mapsto \mathcal{C}_s$ in the conclusion. When v is of sort τ, it is satisfied as a consequence of the first axiom (schema) in $\mathcal{R}$'s specification and the fact that $\alpha(v)$ is a constructor term by Lemma 5.7. With this observation, it is clear that the extended variable assignment satisfies the conclusion. For **Abstract 2**, a similar argument shows that an extended variable assignment which assigns v to the value of $C\,\mathbf{u}$ under $\mathcal{R}, \alpha$ must satisfy the conclusion. For **Abstract 3**, the argument is again similar, but this time we must extend α to map each v_i to the value of $S_C^{(i)}\,u$ under $\mathcal{R}, \alpha$.

(Literal level rules) The case of **Orient** and **Inconsistent** is obvious. For **Remove 1** the claim follows by definition of satisfaction for labeling pairs. For **Remove 2** we rely on the fact that $\mathcal{R}, \alpha$ satisfies $is_C\,v$ exactly when it satisfies $v \mapsto \{C\}$, for any C. This follows from Lemma 5.7 and the first and second axiom schemas.

(Upward closure rules) The claim follows from basic properties of equality.

(Downward closure rules) The result follows from Lemma 5.7 and basic properties of the term algebra $\mathcal{T}(\Omega)$.

(Selector rules) In case of **Instantiate 1** and **2** the claims follow from the definition of satisfaction for labeling pairs, the *Inst* predicate, the first three axiom schemas, and Lemma 5.7. For **Collapse 1** the result follows by the third axiom schema; for **Collapse 2** by the fourth schema, Lemma 5.7 and the definition of satisfaction for labeling pairs.

(Labeling rules) The claim follows by simple Boolean reasoning and the definition of satisfaction for labeling pairs. $\square$

Proposition 11.4 (Soundness). *If a set E_0 has a refutation tree, then it is unsatisfiable in $\mathcal{R}$.*

Proof. By structural induction on refutation trees and the previous lemma. $\square$

To prove completeness we will rely on the next three lemmas. First we need a couple of definitions. If E is a multiset of literals, we write $\sim_E$ for the equivalence relation induced by oriented equations in E. We also define $\mathit{lbls}_E(u)$ as the intersection of all label sets L where $v \mapsto L$ appears in E for some $v \sim_E u$.

Lemma 11.5. *Suppose E is a node in a derivation tree and that E contains an oriented equation of the form $S_C^{(i)}\, u \to v$ for some C (of arity n), u, v, and i, where $1 \leq i \leq n$. We will call this an* oriented selector equation. *Then at least one of the following is also true:*

(i) E also contains an oriented equation of the form $C\,\mathbf{w} \to u'$ for some $\mathbf{w}$ and u' where $u' \sim_E u$.

(ii) $C \notin \mathit{lbls}_E(u)$

(iii) There exist $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$ such that for each $1 \leq k \leq n$, $S_C^{(k)}\, u_k \to v_k \in E$ and $u_k \sim_E u$.

Proof. The proof is by induction on derivation trees. The base case is trivial since the root of a derivation tree has no oriented equations. For the inductive case, we consider each of the rules. First note that if a rule does not introduce, change, or delete any oriented selector equations and furthermore does not delete or change any oriented equations of the form $C\,\mathbf{w} \rightarrow u'$, then the property is trivially preserved. This covers the following rules: **Abstract 1**, **Abstract 2**, the literal level rules, **Simplify 1**, **Cycle**, **Instantiate 2**, and the labeling rules. We now consider the others:

Abstract 3. This rule introduces new oriented selector equations. For these, it is easy to see that condition (iii) is satisfied. It is also easy to see that the property is preserved for any other oriented selector equations.

Simplify 2. This rule may change an oriented selector equation from $S_C^{(i)}\,u \rightarrow v$ to $S_C^{(i)}\,u' \rightarrow v$ when $u \rightarrow u'$. However, in this case, we have $u \sim_E u'$, and it follows that the property is preserved.

Superpose. If we have two oriented selector equations: $S_C^{(i)}\,u \rightarrow v$ and $S_C^{(i)}\,u \rightarrow v'$, with $v \succ v'$, then the first of these may be eliminated by the **Superpose** rule. If the eliminated oriented selector equation was needed to fulfill condition (iii) for some other oriented selector equation in the premise, then we must ensure that the property still holds in the conclusion. However, notice that $S_C^{(i)}\,u \rightarrow v'$ may be used instead and so the property holds.

Compose. Suppose $S_C^{(i)}\,u \rightarrow v$ is rewritten to $S_C^{(i)}\,u \rightarrow v'$. It is easy to see that the property holds for the new oriented selector equation for the same reasons as it did for the old. Also, if the old oriented selector equation was used to fulfill condition (iii) for some other oriented selector equation, then the new one does so as well.

Decompose. This rule may eliminate an oriented equation of the form $C\,\mathbf{w} \rightarrow u'$ which might affect condition (i) for some oriented selector equation. However, it only does so when there exists another oriented equation of the form $C\,\mathbf{v} \to u'$ that is not eliminated. This can be used to satisfy condition (i) instead.

Instantiate 1. This rule eliminates oriented selector equations which could affect condition (iii) for some other oriented selector equation. However, it also introduces an oriented equation of the form $C\,\mathbf{w} \to u$, so condition (i) will now apply to such oriented selector equations.

Collapse 1. This rule eliminates an oriented selector equation which could affect condition (iii) for some other oriented selector equation. However, it is easy to see that because we have an oriented equation of the form $C\,\mathbf{w} \to u$, condition (i) must apply to such oriented selector equations.

Collapse 2. This rule eliminates an oriented selector equation which could affect condition (iii) for some other oriented selector equation. However, it is easy to see that because $C \notin \mathit{lbls}_E(u)$, condition (ii) must apply to such oriented selector equations. □

Lemma 11.6. *No irreducible leaf E in a derivation tree contains occurrences of selector symbols.*

Proof. The claim is trivially true if $E = \{\bot\}$, so assume that $E \neq \{\bot\}$. Since E is irreducible, by the rule **Abstract 3** and Lemma 11.1(1), every occurrence of a selector in E must be in an oriented equation of the form $S_C^{(i)}\, u \to v$, for some constructor $C : s_1 \cdots s_n \to \tau$ and an abstraction variable u of sort τ. So assume that $S_C^{(i)}\, u \to v \in E$. By Lemma 11.5, we know that one of three conditions applies. The first case is that condition (i) holds: E also contains an oriented equation of the form $C\,\mathbf{w} \to u'$ for some $\mathbf{w}$ and u' where $u' \sim_E u$. Since E is irreducible, we must have that $u' = u$, but then **Collapse 1** applies, contradicting the irreducibility of E. The second case is (ii): $C \notin \mathit{lbls}_E(u)$. Again, because E is irreducible, this means that E contains $u \mapsto L$ and $C \notin L$. Thus, **Collapse 2** applies, again a contradiction. Finally, the third case is (iii): there exist $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$ such that for each $1 \leq k \leq n$, $S_C^{(k)}\, u_k \to v_k \in E$ and $u_k \sim_E u$. Again, because E is irreducible, we must have that $u_k = u$ for each k. Also, since (ii) does not apply and **Split 1** cannot be applied, E must contain $u \mapsto \{C\}$. But this means that **Instantiate 1** applies, again yielding a contradiction.

□

Lemma 11.7. *Every irreducible leaf E other than $\{\bot\}$ in a derivation tree is satisfiable in $\mathcal{R}$.*

Proof. We build a valuation α of $\mathcal{V}ar(E)$ that satisfies E in $\mathcal{R}$. To start, let

$$\begin{aligned} V &= \{v \mid t \to v \in E \text{ for some } t\} \\ T_v &= \{t \mid t \to v \in E\} \text{ for all } v \in V \end{aligned}$$

Observe that the sets T_u and T_v are disjoint for all distinct u and v, otherwise E would contain two equations of the form $t \to u$ and $t \to v$, and so would be reducible by **Superpose**. Furthermore, for all $v \in V$, T_v contains at most one non-variable term. To see that, recalling that E contains no occurrences of selector symbols by Lemma 11.6, assume that T_v contains a term of the form $C\,\mathbf{u}$. Again by well-sortedness, it is enough to argue that T_v contains no additional terms of the form $C'\,\mathbf{u}'$ of the same sort as v's. But such terms cannot be in T_v. If $C = C'$ then **Decompose** applies. If $C \neq C'$, notice that whenever an oriented equation of the form $C\,\mathbf{u} \to v$ is introduced, we also have $v \mapsto \{C\}$. Since label sets never grow, at some point we have to have had both $v \mapsto \{C\}$ and $v \mapsto \{C'\}$. Since **Refine** must have been applied to these two labeling pairs, E must now contain $v \mapsto \emptyset$ and is thus reducible by **Empty**.

Now consider the relation $\lessdot$ over V defined as follows:

$$u \lessdot v \text{ iff } E \text{ contains an equation of the form } C\,\mathbf{u}u\mathbf{u'} \to v.$$

By the **Cycle** rule and the assumptions on E, the finite relation $\lessdot$ is acyclic and hence well founded. We can define a valuation α of V into $\mathcal{R}$[6] by well founded induction on $\lessdot$.

Let $\{v_1, \ldots, v_n\}$ be the set of all the $\lessdot$-minimal elements of V such that for $i = 1, \ldots, n$, $c_i \to v_i \in E$ with c_i a constant symbol (a nullary constructor). For $i = 1, \ldots, n$ we define $\alpha(v_i) = c_i$. Now let $\{v_{n+1}, \ldots, v_{n+k}\}$ be the remaining $\lessdot$-minimal elements of V. If v_i is of some sort τ, we know by a previous observation that $v_i \mapsto L \in E$. Note that by the **Empty** and the **Split** rules, $C \in L$ for some non-nullary C. Moreover, C must be an infinite constructor, or otherwise an equation of the form $C\,\mathbf{u} \to v_k$ would be in E by **Instantiate 2**, making v_k non-$\lessdot$-minimal. We then define $\alpha(v_k) = C\,t_1 \cdots t_m$ where C is some infinite constructor in L of arity $m > 0$ and $C\,t_1 \cdots t_m$ is some term in $\mathcal{T}(\Omega) \setminus \{\alpha(v_1), \ldots, \alpha(v_{n+k-1})\}$.

We are now left with defining $\alpha(v)$ for all non-minimal $v \in V$. If v is non-minimal, then there must be an equation of the form $C\,u_1 \cdots u_k \to v$ in E for some constructor C. Furthermore, $k \geq 1$ (otherwise v would be minimal) and $u_i \lessdot v$ for all $i = 1, \ldots, k$. We then define $\alpha(v) = C\,\alpha(u_1) \cdots \alpha(u_k)$.

We now show by induction on $\lessdot$ that the valuation α just defined is an injection of V into $\mathcal{T}(\Omega)$. Let u, v be two distinct elements of V of the same sort.

If u and v are both $\lessdot$-minimal in the set $\{v_1, \ldots, v_n\}$ defined earlier, then $\alpha(u) \neq \alpha(v)$ because the sets $T_{v_1}, \ldots, T_{v_n}$ are mutually disjoint. If one (or both) of them is in $\{v_{n+1}, \ldots, v_{n+k}\}$ then $\alpha(u) \neq \alpha(v)$ by construction.

If u, say, is not $\lessdot$-minimal, then both u and v must be of some sort τ. It follows that $\alpha(u), \alpha(v)$ are terms of the form $C\,\alpha(u_1) \cdots \alpha(u_n)$, $C'\,\alpha(v_1) \cdots \alpha(v_{n'})$, respectively, with $n, n' \geq 1$. Now, if $C \neq C'$, then $\alpha(u)$ and $\alpha(v)$ are trivially distinct

[6]Whose universe, recall, is the term algebra $\mathcal{T}(\Omega)$.

terms. If $C = C'$, then $n = n'$; however, $u_i \neq v_i$ for some i otherwise $C\,u_1 \cdots u_n \to u$ and $C\,u_1 \cdots u_n \to v$ would be in E and **Superpose** would apply. If u_i and v_i are distinct then by induction $\alpha(u_i)$ and $\alpha(v_i)$ are distinct, therefore $\alpha(u)$ and $\alpha(v)$ are distinct as well.

Now we can extend α to the whole $\mathcal{V}ar(E)$ by defining it for the remaining (input or abstraction) variables of E. Each such variable x occurs in an equation of the form $x \to v$ in E. Hence we define $\alpha(x) = \alpha(v)$. For later reference, let α' be the homomorphic extension of α to the set of Σ-terms over $\mathcal{V}ar(E)$.

The valuation α satisfies every element e of E. If e has the form $u \not\approx v$ with u, v distinct, then α satisfies e for being injective over the abstraction variables of E. If e has the form $t \to v$, then α satisfies e because $\alpha(v) = \alpha'(t)$ by construction. If e has the form $v \mapsto L$ where v has sort τ consider the following two cases. If $C\,u_1 \cdots u_k \to v \in E$ for some $C\,u_1 \cdots u_k$ then it is not difficult to show that L must be $\{C\}$. But then $\alpha(v) = C\,\alpha(u_1) \cdots \alpha(u_k)$ by construction. If there is no $C\,u_1 \cdots u_k \to v \in E$, then $\alpha(v)$ is defined as some term $C\,t_1 \cdots t_k$ where $C \in L$. In both cases, it is then immediate that α satisfies $v \mapsto L$.

To conclude the proof it is enough to observe that, for being irreducible, E can only contain elements of the forms listed above. $\square$

Proposition 11.8 (Completeness)**.** *If a set E_0 is unsatisfiable in $\mathcal{R}$, then it has a refutation.*

Proof. We prove the contrapositive of the proposition. Assume that E_0 has no refutations. By Proposition 11.2, there is a derivation tree for E_0 with an irreducible leaf $E \neq \{\bot\}$. By Lemma 11.7, E is satisfiable in $\mathcal{R}$. It follows by a repeated application of Lemma 11.3 that E_0 is satisfiable in $\mathcal{R}$ as well. $\square$

12 Strategies and Efficiency

It is not difficult to see that the problem of determining the satisfiability of an arbitrary set of literals is NP-complete. A subset of the problem (a simple case with two constructors) was shown to be NP-hard in [36]. To see that it is in NP, we note that given a type completion, no additional splits are necessary, and the remaining rules can be carried out in polynomial time. However, as with other NP-complete problems (Boolean satisfiability being the most obvious example), the right strategy can make a significant difference in practical efficiency.

12.1 Strategies

A *strategy* is a predetermined methodology for applying the rules. Before discussing our recommended strategy, it is instructive to look at the closest related work. Oppen's original algorithm is roughly equivalent to the following: After abstraction, apply the selector rules to eliminate all instances of selector symbols. Next, apply upward and downward closure rules (the bidirectional closure). As you go, check for conflicts using the rules that can derive $\perp$. We will call this the *basic strategy.* Note that it excludes the splitting rules: because Oppen's algorithm assumes a single constructor, the splitting rules are never used. A generalization of Oppen's algorithm is mentioned in [50]. They add the step of initially guessing a "type completion". To model this, consider the following simple **Split** rule:

$$\textbf{Split} \quad \frac{u \mapsto \{C\} \cup L,\ E}{u \mapsto \{C\},\ E \qquad\qquad u \mapsto L,\ E} \quad \text{if} \quad L \neq \emptyset$$

Now consider a strategy which invokes **Split** greedily (after abstraction) until it no longer applies and then follows the basic strategy. We will call this strategy the *greedy splitting* strategy.

One of the key contributions of this section is to recognize that the greedy splitting strategy can be improved in two significant ways. First, the simple **Split** rule should be replaced with the smarter **Split 1** and **Split 2** rules. Second, these rules should be delayed as long as possible. We call this the *lazy splitting* strategy. The lazy strategy reduces the size of the resulting derivation in two ways. First, notice that **Split 1** is only enabled when some selector is applied to u. By itself, this eliminates many needless case splits. Second, by applying the splitting rules *lazily* (in particular by first applying selector rules), it may be possible to avoid splitting completely in many cases. We already saw in Section 10 that Example 10.2 can be solved using only a single case split, instead of the 2^7 splits required by a naive type completion. Here, we look at another example that emphasizes the advantages of lazy splitting.

Example 12.1. *Suppose we have the following simple tree data type:*

$$tree \ := \ node(left : tree,\ right : tree) \mid leaf;$$

Let leaf be the designated term for both selectors and then consider the following set of literals: $\{left^n(Z) \approx X, is\_node(Z), Z \approx X\}$.

A term graph for Example 12.1 is shown in Figure 11. After applying all available rules except for the splitting rules, the resulting set of literals looks like this:

$$\begin{aligned}\{ \quad & Z \to u_0, X \to u_0, u_0 \mapsto \{node\}, node(u_1, v_1) \to u_0, u_n \to u_0, \\ & left(u_1) \to u_2, \ldots, left(u_{n-1}) \to u_n, u_1 \mapsto \{leaf, node\}, \ldots, u_n \mapsto \{leaf, node\}, \\ & right(u_1) \to v_2, \ldots, right(u_{n-1}) \to v_n, v_1 \mapsto \{leaf, node\}, \ldots, v_n \mapsto \{leaf, node\}\},\end{aligned}$$

Notice that there are $2n$ abstraction variables labeled with two labels each. If we eagerly applied the naive **Split** rule at this point, the derivation tree would reach size $O(2^{2n})$.

$$\begin{tikzcd}
 & & X & & Z \arrow[dl] \arrow[dr] & \\
 & & & \mathit{left}(Z) \arrow[dl] \arrow[dr] & & \mathit{right}(Z) \\
 & & \mathit{left}(\mathit{left}(Z)) \arrow[dl, dotted, no head] & & \mathit{right}(\mathit{left}(Z)) & \\
 & \mathit{left}^{n-1}(Z) \arrow[dl] \arrow[dr] & & & & \\
 \mathit{left}^{n}(Z) & & \mathit{right}(\mathit{left}^{n-1}(Z)) & & &
\end{tikzcd}$$

Figure 11: Term graph for Example 12.1

Suppose, on the other hand, that we use the lazy strategy. First notice that **Split 1** can only be applied to n of the abstraction variables ($u_i, 1 \leq i \leq n$). Thus the more restrictive side-conditions of **Split 1** already reduce the size of the problem to at worst $O(2^n)$ instead of $O(2^{2n})$. However, by only applying it lazily, we do even better: suppose we split on u_i. The result is two branches, one with $u_i \mapsto \{\mathit{node}\}$ and the other with $u_i \mapsto \{\mathit{leaf}\}$. The second branch induces a cascade of (at most n) applications of **Collapse 2** which in turn results in $u_k \mapsto \{\mathit{leaf}\}$ for each $k > i$. This eventually results in $\perp$ via the **Empty** and **Refine** rules. The other branch contains $u_i \mapsto \{\mathit{node}\}$ and results in the application of the **Instantiate 1** rule, but little else, and so we will have to split again, this time on a different u_i. This process will have to be repeated until we have split on all of the u_i. At that point, there will be a cycle from u_0 back to u_0, and so we will derive $\perp$ via the **Cycle** rule.

Because each split only requires at most $O(n)$ rules and there are $n-1$ splits, the total size of the derivation tree will be $O(n^2)$. In fact, if we start at u_{n-1} and work our way down, each split will take only $O(1)$, so the total size of the derivation tree will be $O(n)$.[7]

12.2 Experimental Results

We have implemented both the lazy and the greedy splitting strategies in the theorem prover CVC3 [8]. What is necessary for comparing the two splitting strategies is to have some benchmarks that require non-trivial amounts of splitting. To produce such benchmarks, we randomly generated conjunctions of literals over the mutually recursive algebraic data types *nat*, *list*, and *tree* mentioned in the introduction.

As expected, most of the benchmarks are quite easy. In fact, over half of them are solved without any case splitting at all. However, a few of them did prove to be somewhat challenging, at least in terms of the number of splits required. We tried both the greedy and lazy strategies on all benchmarks and categorized the benchmarks according to how many case splits were required in the worst case by either strategy.

Table 4 shows the results. As expected, for easy benchmarks that don't require many splits, the two algorithms perform almost identically. However, as the difficulty increases, the lazy strategy performs much better. For the hardest benchmarks, the lazy strategy outperforms the greedy strategy by more than an order of magnitude. Notice that the disparity in case splits is even greater: for nontrivial benchmarks, the number of case splits taken by the lazy strategy is always much less than that taken by the greedy strategy: over two orders of magnitude for the hardest benchmarks.

[7]This does not mean the total time is necessarily $O(n)$. In general, processing a node includes bidirectional closure and checking for cycles which requires $O(n)$ steps (see [37], for example). So the total processing time is bounded by $O(n \cdot m)$, where m is the size of the derivation tree. In this case, the total time is bounded by $O(n^2)$.

Worst Case Splits	Num. of Tests	Sat	Unsat	Greedy Splits	Greedy Time (s)	Lazy Splits	Lazy Time (s)
0	4416	306	4110	0	24.6	0	24.6
1-5	2520	2216	304	6887	16.8	2414	17.0
6-10	692	571	121	4967	5.8	1597	5.7
11-20	178	112	66	2422	2.3	517	1.6
21-100	145	73	72	6326	4.5	334	1.1
101+	49	11	38	16593	9.8	73	0.3

Table 4: Greedy vs. Lazy Splitting

Part IV

Theorem Prover over Algebraic Data Structures

13 Introduction and Motivation

In part IV we describe a concept theorem prover based on type theory, called **Term Builder**, which has been developed to support reasoning about functions over algebraic types. Nonetheless, one can define much more involved theories and concepts, and interactively prove theorems about them as well.

From the start of this concept project, we have been motivated by the idea to base it entirely on the Curry-Howard Isomorphism, known as "Programs are Proofs, Propositions are Types". For an exposition of this paradigm, see references [16] and [45]. Among the reasons for our choice of formalism, is the fact that the language of programs and the language of proofs are the same, and can be mixed in every way. Since proof checking coincides with type checking, type annotations can be put into proofs without changing the syntax, just as the logical information can be embedded in programs seamlessly. This lets us avoid having to program the same things twice. For example, the following declarations carry no formal distinction to the prover. Each one introduces a constant of a given type:

Set	:	Type	declaration of a user defined type;
$axiom1$	:	$f(2) \approx f(3)$	postulating an assertion;
$append$	:	$List \to List \to List$	declaration of a function;

Another motivation has been to develop a clear, convenient, and intuitive user interface. By now, the notion of Pure Type Systems (PTS, reference [18]) has become standard, when talking about type systems. It continues to be a reference frame, relative to which other type systems are being positioned, or compared. Introduced by Henk Barendregt in [4], the λ-cube is a particular family of PTS, where the calculus of constructions, CC, is the most expressive one. Among even more powerful type systems are the calculus of constructions with universes $CC\omega$ (Alexandre Miquel, [34]), which is an adaptation of the *Extended Calculus of Constructions* **ECC** (Luo [30]) and the calculus of inductive constructions **CIC**. The expressiveness of our type system is positioned between CC and **CIC**.

Our prover is not intended to be industrial strength, as it has very little automation, and even proofs of relatively simple statements require too much time. Also, there are no built-in libraries of any kind, other than what has been proved and saved by the user. Our type rules are borrowed sequentially from the subsystems of **CIC**, in this order:

$$\lambda^{\to} \quad \to \quad \textbf{System F} \quad \to \quad \lambda P2 \quad \to \quad CC$$

In our formalism, reductions are less essential since they carry computational meaning. Instead, we concentrate on equations, as they constitute propositions.

Therefore, we view the programming language capabilities of **Term Builder** not as a way to run programs, but as the means to define functions. This is what distinguishes **Reductionism** from **Equationism**.

The theme of reductionism and equationism can be found already in early, as well as modern type theory, the λ-cube, and the proof verifier Coq. Essentially, if two terms t_1 and t_2 reduce to the same normal form, it constitutes a proof of their equality. This means, one has to make the proof checker step out into the meta realm, the realm of execution of programs, rather than establish $(t_1 \approx t_2)$ by finding an explicit proof term $p : (t_1 \approx t_2)$. Here again we opt for the explicit proof term of an equality, even if it is in some sense isomorphic to the execution trace, that is, they can be put into a piece by piece correspondence with each other. One way in which our approach takes the equationist point of view, is by rejecting the conversion rule, common in type systems. This rule is discussed in section 15.4.1. By avoiding this rule (at the expense of adding some others) we do not rely on any computation on propositions, but always treat them as static structures during proof-checking.

This idea of *Replacement of Proof by Computation* is sometimes referred to as the Poincaré Principle. This issue is thoroughly examined by Henri Poincaré in reference [9]. In his essay "On the nature of mathematical reasoning" he analyses an example which establishes $2 + 2 = 4$. After the necessary arithmetic definitions and calculations, he writes: *It cannot be denied that this reasoning is purely analytical. But if we ask a mathematician, he will reply: "This is not a demonstration properly so called, it is a verification". We have confined ourselves to bringing together one or two purely conventional definitions, and we have verified their identity; nothing new has been learned. Verification differs from proof precisely because it is analyti-*

cal, and because it leads to nothing.

The Poincaré Principle has been part of the inspiration for **Term Builder**. Our earliest attempts to build proofs automatically were based on the idea of extracting them from computations. In a sense, this is a backwards reinterpretation of the Poincaré Principle. Here are the details: we take a proposition, that is, an instantiated programmable predicate, and expose it to partial symbolic evaluation, resembling a backwards type-inference procedure. It turns out, that the proof of the proposition originally taken, can be synthesized directly out of the trace of such partial evaluation, once it has stopped. At a later stage of the development of **Term Builder**, every step of such proof building method became a command button on the user screen, which controls the proof construction process. The rest of this document is organized as follows. Section 14 gives technical background, which is necessary for understanding the rest. Section 15 explains the explicit extentions of the standard formalism present in **Term Builder**, in particular, support of algebraic types. Section 16 proves the soundness of **Term Builder**.

14 Theoretic Background

14.1 Preliminary Considerations

What is the essence of interpreting terms as proofs and propositions as types? A lambda-abstraction (a function) $\lambda x : A.\ t^B$ represents a proof of the proposition $(\forall x : A.\ B)$. Furthermore, $(t_1\ t_2)$ is an application of function t_1 to the argument t_2. When such applications are carried out, it is common to use the concept of β-reduction, which looks as follows: $((\lambda x : A.\ t)a) \longrightarrow_\beta t\{a/x\}$. This means, the λ-term $(\lambda x : A.\ t)$ of type $(\forall x : A.\ B)$ will yield a proof $t\{a/x\}$ of $B\{a/x\}$ for every proof a of A. The connection between β-reduction and Modus Ponens will be explained shortly using the Figure 12.

Calculus of constructions (reference [18]), denoted CC, is part of our formalism. The grammar for the pre-terms of the calculus of constructions is given below. This grammar does not distinguish any syntactic categories. This role is delegated to the type rules presented later in full detail. The point of this grammar is to give the shortest possible description of the totality of expressions in CC. Refinement of this rough schema is given later in Table 5.

$$\text{pre terms } t :== x^{(variable)} \mid \lambda x : t.\ t \mid \forall x : t.\ t \mid (t\ t)$$

To illustrate again the use of abstraction and application, consider the following example: Let $f \equiv (\lambda x : A.\ M^B)$. Also let $(a : A)$. Then $(f\ a)$ is an application of function f to a, and $(f\ a) \longrightarrow_\beta M\{a/x\}^{B\{a/x\}}$.

Let us briefly recall the properties of typed λ-calculi, in particular, the dynamic properties that relate computation with logic. First, a term is said to be in *normal form* if it is not β-reducible. That is, it does not contain a β-reducible subexpression (called a redex). *Weak Normalization* of a calculus states that for every typed term there is a reduction sequence leading to a normal form. *Strong Normalization* asserts that any sequence of reductions of a typed term ends with a normal form. Existence of normal forms is essential for logic. Whenever we have Modus Ponens as an intermediate step in our proof, this introduces a potential redex. For instance, $((\lambda x : A.\ M)a)$ reduces to $M\{a/x\}$. This is equivalent to the proof normalization step in Figure 12, where $\mathcal{H}$ and $\mathcal{D}$ are natural deduction style derivations, and $\mathcal{H}$ is parametrized with a sub-derivation.

Normalizability implies that all such uses of Modus Ponens can be eliminated by way of substitutions, as shown above. In turn, this means that all lemmas have

$$\boxed{\dfrac{\dfrac{\dfrac{\mathcal{H}\left(\dfrac{\textit{assume } A}{A}\right)}{B}}{A \to B}\,(\textit{discharge } A) \qquad \dfrac{\mathcal{D}}{A}}{B} \quad\longrightarrow\quad \dfrac{\mathcal{H}\left(\dfrac{\mathcal{D}}{A}\right)}{B}}$$

Figure 12: Modus Ponens as β-reduction

been expanded and we end up with a proof from first principles. This phenomenon is also related to Gentzen's *Cut Elimination* (reference [20], pages 105-112), except it has been originally carried out for the Sequent Calculus.

14.2 Overview of Dependent Type Theory

We assume the reader is acquainted with natural deduction, some type theory, and its relation to mathematical logic, which is covered in references [16] and [45]. First, we consider the grammatic categories that we shall encounter. They are called terms, types, kinds, and type universes $(\mathsf{Type}_i)_{i=1}^{\infty}$. Here is the lineup, with the typing of each entity by the next one.

$$(\textit{a term}) : (\textit{a Type}) : (\textit{a Kind}) : (\textit{Universe}\ \mathsf{Type}_1) : (\textit{Universe}\ \mathsf{Type}_2) : \ldots$$

The constant Type is one of the "kinds" in CC, and $\square$ is the only universe in CC. So the following sequence is an example of the above scheme:

$((\lambda x : X.\ x) : (X \to X) : \mathsf{Type} : \square)$, where $\square$ has no type and ends the chain.

This setup with the single universe $\square$ is what we use in **Term Builder**. The following table shows which syntactic categories play a role in which systems:

Terms t	$\in$	$\lambda^{\to}$	**System F**	$\lambda P2$	CC	$CC\omega$	**CIC**
Types T	$\in$	$\lambda^{\to}$	**System F**	$\lambda P2$	CC	$CC\omega$	**CIC**
Kinds K	$\in$			$\lambda P2$	CC	$CC\omega$	**CIC**
Universes U	$\in$					$CC\omega$	**CIC**

	Functions	Types T	Applications	Comment
$\lambda^{\rightarrow}$	$\lambda x : T.\, t$	$T \rightarrow T$ $\perp$	$(t\ t)$	plain implication falsehood
F	$\lambda x : T.\, t$ $\Lambda X.\, t$	$T \rightarrow T$ $\Pi X.\, T$	$(t\ t)$ $(t\ T)$	plain implication parametric polymorphism
$\lambda P2$	$\lambda x : T.\, t$ $\lambda x : \mathsf{Type}.\, t$ $\lambda x : T.\, T$	$\forall x : T.\, T$ $\forall X : \mathsf{Type}.\, T$ $T \rightarrow \mathsf{Type}$	$(t\ t)$ $(t\ T)$ $(t\ t)$	dependent types parametric polymorphism predicates
CC	$\lambda x : T.\, t$ $\lambda x : \mathsf{Type}.\, t$ $\lambda x : T.\, T$ $\lambda x : \mathsf{Type}.\, T$	$\forall x : T.\, T$ $\forall X : \mathsf{Type}.\, T$ $T \rightarrow \mathsf{Type}$ $\mathsf{Type} \rightarrow \mathsf{Type}$	$(t\ t)$ $(t\ T)$ $(t\ t)$ $(t\ T)$	dependent types parametric polymorphism predicates type transformers

Table 5: Features of Type Systems

For every system, assume that there are *term variables* x and *type variables* X. Traditionally, in **System F** there is no "kinds" level, so the quantification over types carries no domain of quantification, but uses Λ, instead of λ. Therefore, expressions $(\Lambda X.\, t)$ and $(\lambda X : \mathsf{Type}.\, t)$ are equivalent. The same is true for $(\Pi X.\, T)$ and $(\forall X : \mathsf{Type}.\, T)$. Now let us give further intuition about Table 5. In system $\lambda^{\rightarrow}$, the type $\perp$ is given explicitly, since otherwise the propositional logic basis of $\lambda^{\rightarrow}$ would be incomplete. In contrast, in **System F** and beyond, there is a type, equivalent to falsehood, namely $\Pi A.\, A$, or equivalently, $\forall A : \mathsf{Type}.\, A$. This term is supposed to be empty due to the logical consistency requirement, which states that not every proposition is provable. The domain of types (denoted Type) is an essential constant within the framework of CC. Functions mapping data objects into this domain are considered predicates. Recall that in our formalism propositions are types, so by mapping objects into the domain of types, we express their properties,

i. e. : let $a : A$, $P : A \to \mathsf{Type}$. Then (Pa) expresses property P of object a.

14.3 Dependent Types in the Systems of the Lambda Cube

In this section we describe the type rules, which we have taken from the calculus of constructions to be integrated with the logic of **Term Builder**. They include Modus Ponens (the dependent version), and the four dependency rules for the formation of universally quantified types. To illustrate the dependencies across types and terms, we first focus on the standard type rules which are taken from the progression of systems considered below ($\lambda^{\to}$, **System F**, $\lambda P2$, CC). This progression is cumulative: each rule presented for one system carries over to the next one. We start with two basic rules, and the rest is filled in by considering the sequence of those systems. The grammatic categories are as follows: f is a term, a is a term or a type, A is a type or kind, and B is also a type or kind.

$$\frac{\text{(axiom)}}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma \vdash f : (\forall x : A.\ B) \quad \Gamma \vdash a : A}{\Gamma \vdash (f\ a) : B\{a/x\}} \ \text{(Modus Ponens)}$$

Now we can give the incremental additions to the type system, corresponding to the progression: $\lambda^{\to}$, **System F**, $\lambda P2$, CC. Later we will see that the four dependency rules presented here can be compacted into one general rule serving to form arbitrary λ-abstractions, and to generate universally quantified types.

14.3.1 Simply Typed Lambda Calculus ($\lambda^{\to}$)

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash A, B : \mathsf{Type} \qquad \text{(Terms dependent on Terms, } \lambda^{\to}\text{)}}{\Gamma \vdash (\lambda x : A.\ M) : (A \to B)}$$

This is a standard rule capturing substitutions of terms into terms. For an example of using this rule, let $\Gamma, x : X \vdash x : X$, then $\Gamma \vdash (\lambda x : X.\ x) : (X \to X)$. For simply typed calculus $\lambda^{\rightarrow}$, the standard arrow type $(A \rightarrow B)$ is sufficient, and serves as abbreviation for $(\forall x : A.\ B)$, when $x \notin FV(B)$ in more expressive systems. To see this, it is sufficient to look at Table 5 and notice that for $\lambda^{\rightarrow}$ the type of a λ-abstraction is an implication, but in $\lambda P2$ the type already involves the universal quantifier. In systems considered next, $A \rightarrow B$ must be replaced by $(\forall x : A.\ B)$. This is a product type, whose intended meaning is a function space where the range of each function depends on its argument.

14.3.2 System F

$$\frac{\Gamma \vdash T : \mathsf{Type} \qquad \Gamma, X : \mathsf{Type} \vdash M : T \qquad \text{(Terms dependent on Types, } \mathbf{F}\text{)}}{\Gamma \vdash (\lambda X : \mathsf{Type}.\ M) : (\forall X : \mathsf{Type}.\ T)}$$

This rule gives us the second order type theory **System F**, since it allows quantification over types, and types may appear as arguments to functions. This is the origin of the term "parametric polymorphism". We can write, say, an identity function over any type as follows: $(\lambda X : \mathsf{Type}.\ \lambda x : X.\ x)$. Its type would be $(\forall X : \mathsf{Type}.\ X \rightarrow X)$, which also serves as a replacement for proposition "**true**". It is useful to recall J.-Y. Girard's account on the naive interpretation of the quantification over types. Namely, such interpretation says that an object of type $\Lambda X.V$ assigns an object of type $V\{U/X\}$ for every type U. We reproduce his account here from reference [20], page 83: *"This interpretation runs up against a problem of size: in order to understand $\Pi X.V$, it is necessary to know all the $V\{U/X\}$. But among all the $V\{U/X\}$ there are some which are (in general) more complex than the type which we seek to model, for example $V\{\Pi X.V/X\}$. So there is a circularity in the naive interpretation, and one can expect the worst to happen. In fact, it all works out, but the system is very sensitive to modifications, which are not of logical nature"*. An example of such sensitivity is a quote by Thierry Coquand from his work [14], page 19: *"J.-Y. Girard told me that his results on the system U must* *show that the calculus of constructions with four levels is inconsistent".*

14.3.3 Second Order Dependent Type Theory ($\lambda P2$)

$$\frac{\Gamma \vdash T : \mathsf{Type} \qquad \Gamma, x : T \vdash M : \mathsf{Type} \qquad \text{(Types dependent on Terms, } \lambda P2\text{)}}{\Gamma \vdash (\lambda x : T.\ M) : (T \to \mathsf{Type})}$$

This rule is significant from the logical point of view, since it gives us the notion of predicates. When type expressions become instantiated with data objects, what we get is concrete assertions. For example, if $A : \mathsf{Type}$, then $P : A \to \mathsf{Type}$ is a predicate, and $P(a^A)$ is an assertion about a. Here we step outside of **System F**, which does not have dependent types. An example of using a predicate within a term is $(\lambda X : \mathsf{Type}.\ \lambda P : X \to \mathsf{Type}.\ \lambda x : X.\ (P\ x))$.

14.3.4 Calculus of Constructions *CC*

$$\frac{\Gamma, X : \mathsf{Type} \vdash M : \mathsf{Type} \qquad \text{(Types dependent on Types, } CC\text{)}}{\Gamma \vdash (\lambda X : \mathsf{Type}.\ M) : (\mathsf{Type} \to \mathsf{Type})}$$

Using this last dependency rule, one has the power to construct new types out of old types. For example, let T be a type, and declare $List : \mathsf{Type} \to \mathsf{Type}$. Then $(List\ T)$ is the type of lists with elements of type T. Otherwise, we would have to create a new list type for every element type. This problem has a workaround, for example in C++, via templates. Note that we have only used $\forall$ and Type as our logical primitives. We follow the standard technique to simplify matters, in particular, every logical connector can be encoded in second order type theory **System F**. This encoding is presented in Table 6. It is due to J.-Y. Girard, and appears in reference [20].

$\bot$	$\equiv$	$(\forall X : \mathsf{Type}.\ X)$
$A \to B$	$\equiv$	$\forall x : A.\ B$
$\neg A$	$\equiv$	$(A \to \bot)$
$A \wedge B$	$\equiv$	$\forall X : \mathsf{Type}.\ (A \to B \to X) \to X$
$A \vee B$	$\equiv$	$\forall X : \mathsf{Type}.\ (A \to X) \to (B \to X) \to X$
$\exists x : A.\ B$	$\equiv$	$\forall X : \mathsf{Type}.\ (\forall x : A.\ B \to X) \to X$

Table 6: Translation of $\{\bot, \neg, \vee, \wedge, \to, \forall, \exists\}$ into $\{\forall, \mathsf{Type}\}$

14.4 Calculus of Algebraic Constructions

14.4.1 The Need for Dependent Case Analysis

It is well known that analysis by cases is representable in **System F** (reference [20], pages 84-85). This applies if the branches of the conditional statement have equal return types. However, a well recognized difficulty arises if we try to define dependent case analysis where the type of each branch is a predicate, parametrized with the argument that is analyzed by cases. The difficulty is that selection of data fields from constructor terms is only possible "by values", and cannot be done in constant time (references [15], [20] page 91, [48]). Therefore, standard pattern matching is impossible to represent internally. An illustration of the solution for this situation is presented by the following example. Let $N ::= \{0 \mid S\ N\}$. This denotes that a member of N is either the constant 0 or successor of another member of N. The rule introducing the **case** operator for the type N looks as follows:

$$\frac{\Gamma \vdash t : N \qquad \Gamma \vdash M_1 : (P\ 0) \qquad \Gamma,\ m : N \vdash M_2 : P(Sm)}{\Gamma \vdash (\mathbf{case}\ (t) \mid of\ 0\ \mapsto M_1 \mid of\ S\ m\ \mapsto M_2) : (P\ t)}$$

The main point of interest is that this rule combines two cases. The first branch, when $t \equiv 0$, returns a proof of the property P parametrized with the numeral 0. The second branch, when $t \equiv (Sm)$, returns a proof of the same property P for a numeral (Sm). Altogether, based on this exhaustive combination of cases (two, in this case) we are assured that P holds for all numerals, hence we are able to derive the proposition $(P\ t)$ for any $t : N$. Such dependent elimination principles are accepted explicitly in the calculus of inductive constructions **CIC**. In the next subsection we will present the restriction of **CIC** to the calculus with algebraic types.

14.4.2 Adaptation of the Calculus of Inductive Constructions

Here we present a fragment of the calculus of inductive constructions **CIC** that is relevant for our purposes. We do not use the full power of the calculus of inductive constructions, but only what we will call "calculus of algebraic constructions". It includes a way to define an algebraic type, a type rule for dependent case analysis, and conditional evaluation rule. We now denote the domains of types and kinds in **CIC** by Type' and $\Box'$ respectively, to make a syntactic distinction from Type and $\Box$. Figure 13 presents the relevant fragment of **CIC**. The following declaration is a particular case of an inductive type declaration for the calculus of inductive constructions, as outlined in reference [39]:

$$\begin{array}{l} \mathsf{inductive}\ I : \mathsf{Type}\ \mathsf{with} \\ \qquad C_1 : (X_{1,1} \times \ldots \times X_{1,k_1}) \to I \\ \qquad \ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots \\ \qquad C_n : (X_{n,1} \times \ldots \times X_{n,k_n}) \to I \\ \mathsf{end} \end{array}$$

This declaration consists of the name of our algebraic type I, the constructor names C_i and their argument types $X_{i,j}$. The rule below is the dependent case analysis rule from the calculus of inductive constructions, as specified in reference [5]:

$$\frac{\Gamma \vdash P : I \to \mathsf{Type} \quad \Gamma \vdash t : I \quad \{\Gamma, \{x_{i,j} : X_{i,j}\}_{j=1}^{k_i} \vdash t_i : (P\ C_i\mathbf{x_i})\}_{i=1}^{n}}{\Gamma \vdash \mathsf{match}\ t\ \mathsf{in}\ I\ \mathsf{return}\ P\ \mathsf{with}\ \{C_i\mathbf{x_i} \mapsto t_i\}_{i=1}^{n} : (P\ t)}$$

$$\frac{}{\Gamma, x : T \vdash x : T} \qquad \frac{}{\Gamma \vdash \mathsf{Type}' : \Box'} \qquad \frac{\Gamma \vdash M : A \quad (A \longleftrightarrow_{\beta\iota} B)}{\Gamma \vdash M : B \quad (\mathsf{conv})}$$

$$\frac{\Gamma \vdash T : \mathcal{D}om \qquad \Gamma, x : T \vdash U : \mathsf{Type}'}{\Gamma \vdash (\forall x : T.U) \ : \ \mathsf{Type}' \quad (\mathsf{all})_1} \qquad \frac{\Gamma \vdash T : \mathcal{D}om \qquad \Gamma, x : T \vdash U : \Box'}{\Gamma \vdash (\forall x : T.U) \ : \ \Box' \quad (\mathsf{all})_2}$$

$$\frac{\Gamma, \ x : T \vdash M : U \quad \Gamma \vdash (\forall x : T.U) : \mathcal{D}om}{\Gamma \vdash (\lambda x : T.M) : (\forall x : T.U) \quad (\mathsf{abs})} \qquad \frac{\Gamma \vdash M : (\forall x : T.U) \ \ \Gamma \vdash N : T}{\Gamma \vdash (M \ N) : U\{N/x\} \quad (\mathsf{app})}$$

where $\mathcal{D}om \in \{\mathsf{Type}', \Box'\}$ and $(\forall x : T.U) \equiv (T \to U)$, if $x \notin FV(U)$

$$\frac{\Gamma \vdash t : I \quad \Gamma \vdash P : I \to \mathsf{Type}' \quad \{\Gamma, \ \{x_{i,j} : X_{i,j}\}_{j=1}^{k_i} \vdash t_i(\mathbf{x_i}) : (P \ (C_i\mathbf{x_i}))\}_{i=1}^{n}}{\Gamma \vdash \mathsf{match} \ (t) \ \mathsf{in} \ I \ \mathsf{return} \ P \ \mathsf{with} \ \{(C_i\mathbf{x_i}) \mapsto t_i\}_{i=1}^{n} : (P \ t) \qquad (\mathsf{case})}$$

where $I ::= \{C_1 \ X_{1,1} \ldots X_{1,k_1} \mid \ \ldots \ \mid C_n \ X_{n,1} \ldots X_{n,k_n}\}$;

Reductions:

$(beta) : (\lambda x : T. \ M)N \longrightarrow_\beta M\{N/x\}$
$(iota) : \mathsf{match} \ (C_j\mathbf{a}) \ \mathsf{in} \ I \ \mathsf{return} \ P \ \mathsf{with} \ \{C_i\mathbf{x_i} \mapsto t_i\}_{i=1}^{n} \longrightarrow_\iota t_j\{\mathbf{a}/\mathbf{x_j}\}$

Convertibility:

Let $(\longleftrightarrow_{\beta\iota})$ be the reflexive transitive symmetric closure of of the combination of $(\longrightarrow_\beta)$ and $(\longrightarrow_\iota)$. For a term or a type $B(x)$ it is said that $B\{t_1\} \longleftrightarrow_{\beta\iota} B\{t_2\}$, whenever $t_1 \longleftrightarrow_{\beta\iota} t_2$.

Figure 13: Relevant Fragment of the Calculus of Inductive Constructions

The reduction below is a standard evaluation rule for the match clause in the calculus of inductive constructions from [5]:

$$\mathsf{match} \ (C_j\mathbf{a}) \ \mathsf{in} \ I \ \mathsf{return} \ P \ \mathsf{with} \ \{C_i\mathbf{x_i} \mapsto t_i\}_{i=1}^{n} \longrightarrow_\iota t_j\{\mathbf{a}/\mathbf{x_j}\}$$

The meaning of this reduction is as follows. When the argument's top constructor is C_j, it gets matched with the appropriate branch t_j of the match clause. The data vector $\mathbf{a}$ is then plugged into t_j, which becomes the return value $t_j\{\mathbf{a}/\mathbf{x_j}\}$.

15 Type System of Term Builder

15.1 High Level Overview

Our calculus is a restriction of **CIC** to algebraic data types. For any given algebraic data type there is a conditional **case** statement. Such **case** statements are used as branching devices in our language of programs, which is the same as case analysis in the language of proofs. Algebraic types are presented in part II of this thesis, and also in [28]. The difference between "algebraic" and "inductive" is that the constructors of algebraic types can only take arguments of algebraic types, while the restriction for inductive types is not so strong. For example, a constructor of an algebraic type C is not allowed to have an argument of a composite type, say, a functional type $(S \rightarrow C)$. This is why the ground terms of algebraic data types form *free term algebras*.

The overall idea behind the **Term Builder** type system is to borrow most features from the calculus of constructions (including impredicativity of the sort Type, and introduction and elimination rules for λ-abstractions). Another aim is to free ourselves from the conversion rule. The latter part is done by introducing equality as a primitive predicate along with two equality formation rules: (beta) and (eval). Also, there is one equality elimination rule (leq), which treats a given equality proposition as Leibniz equality. For example, if we are given the proposition $(a_1 \approx_A a_2)$, we can infer an implication $(P\ a_1) \rightarrow (P\ a_2)$ for any predicate $(P : A \rightarrow \mathsf{Type})$. It is shown in [20] that algebraic data types can be encoded within **System F**. We prefer a more direct representation, namely, explicit declarations like in a functional programming language, for example, "Haskell".

This section is structured as follows. First we give explicit notation for algebraic types and their constructors. Then we give an example using case analysis and structural induction together, to illustrate general case analysis for algebraic types. Later we turn our attention to the built-in polymorphic equality and motivate our rejection of the conversion rule. This leads us into an analysis of our rules (beta), (eval), (leq), which shows how we are able to avoid conversion.

15.2 Well Founded Recursion and Induction

A standard method to avoid non-well-foundedness of recursion or induction is to generate custom terminating recursors for each inductive data type, like recursor R for natural numbers from Gödel's System T. (see references [20] pages 47-53, [48] page 342). To simplify matters, we opt for explicit unbounded recursion. Naturally, if not properly applied, it may produce unsound proofs, therefore, when we use **Term Builder**, we must take into account appropriate restrictions that will guarantee well-foundedness, but are not part of the formal type rules. Below is the type rule for recursive expressions. It works equally well for programs that use recursion, and for proofs that use induction. We have purposely restricted the type of f to be the function type here, since we would only like to consider recursion in functions or induction in theorems. Let $F = (\forall x : A.B)$,

$$\frac{\Gamma,\ f : F \vdash M : F}{\Gamma \vdash (fix\ f : F.\ M) : F}\ (\mathsf{fix})$$

For this rule to be sound logically, we require that all recursive calls to function f within M have an argument, which is an immediate subterm of the original argument to f. This constitutes our "immediate-subterm induction" – the analogue of primitive recursion in programming. To give an example by what we mean by *immediate subterm* consider the factorial function:

$$f = fix\ f : N \to N.\ \textbf{case}\ n \mid of\ 0\ \mapsto 1 \mid of\ (Sm)\ \mapsto\ n \times f(m)$$

Since the recursion is on the variable n, and m is instantiated by an immediate subterm of n, function f always terminates, so the recursion is well-founded, and the induction is sound.

We have just seen a recursive program, using the functionality of **case**. How do we use the functionality of **case** analysis to enable inductive proofs? Assume $P : (N \to \mathsf{Type})$ is a predicate, and we want to establish $\forall n : N.\ (P\ n)$. The induction principle Ind_N along with its proof ind_N, is given here:

$$Ind_N \ \equiv\ (P\ 0) \to (\forall k : N.\ (P\ k) \to (P\ (Sk))) \to \forall n : N.\ (P\ n)$$

$$\begin{aligned} &ind_N = \lambda p : (P\ 0).\ \lambda\gamma : (\forall k : N.\ (P\ k) \to (P\ (Sk))).\\ &\quad fix\ f : (\forall n : N.\ (P\,n)).\ \lambda n : N.\ \langle P\rangle\ \mathbf{case}\,n\ \mid of\ 0\ \mapsto\ p \mid of\ S\,m\ \mapsto\ (\gamma\,m\,(f\ m))\end{aligned}$$

The term ind_N serves as an inductive proof of the proposition Ind_N, and also illustrates how case analysis of algebraic arguments can be integrated into a proof by induction. The fact that primitive recursion "realizes" induction is examined in reference [39].

15.3 Dependent Typing of the Conditional Operator

Let us extrapolate the (case) rule that we have seen in subsection 14.4.1 onto the general algebraic type C.

$$\frac{\Gamma \vdash t : C \qquad \Gamma \vdash P : C \to \mathsf{Type} \qquad \{\Gamma,\ \{x_{i,j} : X_{i,j}\}_{j=1}^{k_i} \vdash M_i(\mathbf{x_i}) : P(C_i\ \mathbf{x_i})\}_{i=1}^{n}}{\Gamma \vdash (\langle P\rangle\ \mathbf{case}\ (t)\ of\ C_1\ ... \mapsto M_1(\mathbf{x_1}) \mid ... \mid\ of\ C_n\ ... \mapsto M_n(\mathbf{x_n}) : (P\ t)}\ (\mathsf{case})$$

How can this be understood as reasoning by cases? Suppose we are given the premises of the rule (case), namely the set:

$$\{\Gamma, \{x_{i,j} : X_{i,j}\}_{j=1}^{k_i} \vdash M_i(\mathbf{x_i}) : P(C_i\ \mathbf{x_i})\}_{i=1}^{n}$$

This is a complete set with respect to n, that is, a proof term M_i is present for each constructor $C_i \in \{C_i\}_{i=1}^{n}$. Moreover, each $M_i(\mathbf{x_i})$ proves that the predicate P is valid over any instantiation of the variables $\{x_{i,j}\}$. Naturally, if $\Gamma \vdash t : C$, then t has the form $(C_i\ v_{i,1} \ldots v_{i,k_i})$ for some i, and therefore, $M_i\mathbf{v_i}$ proves the proposition $P(C_i\mathbf{v_i})$. The predicate P, in which we are interested, has to be supplied as a syntactic part of the conclusion of the rule (case). This is done by a prefix $\langle \lambda n : C.\ P(n)\rangle$, or, equivalently $\langle P\rangle$, like in Coq. Without such explicit declaration it is not necessarily always possible to infer it from the proof terms $\{M_i\}_{i=1}^{n}$.

Our branches $\{of\ \ (C_i\mathbf{x_i}) \mapsto M_i\}_{i=1}^{n}$ cover each one of the n constructors. Based on that, each is assigned a dependent type corresponding to the constructor. For instance, let the **case** expression be:

$$\langle P\rangle\ \mathbf{case}\ (t)\ \{\ of\ \ (C_i\mathbf{x_i}) \mapsto M_i\}_{i=1}^{n}$$

Then by letting $t \equiv C_j\mathbf{v_j}$, the resulting type is the proposition $(P\ (C_j\mathbf{v_j}))$. Moreover, the proof of this proposition is $M_j\{\mathbf{v_j}/\mathbf{x_j}\}$. Since all constructors of type C are exhausted in this manner, we are sure that $(P\ t)$ holds for any $t : C$.

15.4 Polymorphic Equality Predicate

15.4.1 Conversion Rule

Among other things, our approach differs from others in one important way. We discard the conversion rule (shown below), which is considered to be part of $\lambda P2$, CC, and **CIC** (see references [4], [1] pages 81-86, [49], [38]).

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A, B : \mathcal{D}om \qquad (A \longleftrightarrow_{\beta\iota} B)}{\Gamma \vdash M : B \qquad (\mathsf{conv})} \quad \mathcal{D}om \in \{\mathsf{Type}, \Box\}$$

However, we have an alternative, consistent with our viewpoint, which uses equality explicitly, as explained below. One of the problems with the conversion rule is that it introduces strange polymorphism, which is sometimes exactly what we are trying to avoid: it does not distinguish between "live code" and static objects of discourse. If we want to prove $P(3+5)$, it is not the same as proving $P(8)$, since "$3+5$" should be a structural constant inside the type. This is especially visible when we try to prove $3+5 =_N 8$, and it gets reduced to $8 =_N 8$, which is not what we are proving. However, if we wrote a live program $3+5$, of course it should normalize to 8. Other known approaches to type systems not modulo conversion are [19, 44, 40].

15.4.2 Rule for Polymorphic Equality

As an additional primitive construct we have a built-in polymorphic equality predicate, and introduction and elimination rules for it. In general, equality is used not only for algebraic types, but any typed entities. The type rule for the equality predicate itself is given here:

$$\frac{\Gamma \vdash A : \mathcal{D}om \qquad \mathcal{D}om \in \{\mathsf{Type}, \Box\}}{\Gamma \vdash (\approx_A) : A \to A \to \mathsf{Type}}$$

Note that this captures not only equality of terms, but also equality over the "kinds" level, that is, equality of types. This is coherent with the conversion rule, which works for conversions of kinds in addition to conversion of types.

15.4.3 Beta Conversion

From an equationist point of view, we would like to be able to express and prove statements of the form $(\lambda x.M)N \approx M\{N/x\}$. For such occasions, the following rule gives a simple equational solution:

$$\frac{\Gamma,\ x : A \vdash M : B \qquad\qquad \Gamma \vdash N : A}{\Gamma \vdash \mathit{Beta}(\lambda x : A.M, N) : (M\{N/x\} \approx_B (\lambda x : A.M)N)}\ \textsf{(beta)}$$

This rule enables us to construct proofs of equality of terms before and after β-reduction.

15.4.4 Conditional Expression Reduction

Let us again use type N with zero and successor. Consider this informal example:

$$(\mathbf{case}\ 17 \mid \mathit{of}\ 0\ \mapsto p \mid \mathit{of}\ (Sm) \mapsto f(m)).$$

According to the intended meaning, we would like to be able to derive the equality:

$$(\mathbf{case}\ (17) \mid \mathit{of}\ 0\ \mapsto p \mid \mathit{of}\ (Sm)\ \mapsto f(m)) \approx f(16).$$

More generally, let $\mathbf{x_i}$ represent the vector of data variables, which are formal parameters of the constructor C_i. Also let $\mathbf{v_i}$ represent the vector of data objects, such that $\Gamma \vdash (C_i\mathbf{v_i}) : C$. Now let us have the following syntactic equivalence agreements:

$$CASE_j \equiv \langle P\rangle\ \mathbf{case}\ (C_j\mathbf{v_j})\ \{\ \mathit{of}\ (C_i\mathbf{x_i}) \mapsto M_i\}_{i=1}^{n}$$

Then we are in a position to perform the following derivation, where $\mathit{Eval}(CASE_j)$ is the explicit proof term:

$$\frac{\Gamma \vdash (\langle P\rangle\ \mathbf{case}\ (C_j\mathbf{v_j})\ \{\ \mid \mathit{of}\ (C_i\mathbf{x_i})\ \mapsto M_i\}_{i=1}^{n}) : P(C_j\mathbf{v_j})}{\Gamma \vdash \mathit{Eval}(CASE_j) : (CASE_j \approx_{P(C_j\mathbf{v_j})} M_j\{\mathbf{v_j}/\mathbf{x_j}\})}\ \textsf{(eval)}$$

Our example now looks like this:

$$\frac{\Gamma\ \vdash \overbrace{\langle P\rangle\ \mathbf{case}\ 17 \mid \mathit{of}\ 0\ \mapsto p \mid \mathit{of}\ Sm\ \mapsto\ f(m)}^{CASE_2} : P(17)}{\Gamma\ \vdash \mathit{Eval}(CASE_2) : (CASE_2) \approx f(16)}$$

15.4.5 Leibniz Equality and Equation Elimination Rule

There is a standard approach to equality, namely, the observable equivalence. This is called “Leibniz Equality”. It amounts to saying that if two objects are equal, this means that they satisfy exactly the same set of predicates. For example, in reference [3], Leibniz Equality is defined as the following predicate:

$$\lambda A : \mathsf{Type}.\ \lambda x : A.\ \lambda y : A.\ \forall P : A \to \mathsf{Type}.\ (Py) \to (Px)$$

Our rules (beta), (eval), (leq) fit together to compensate for the absence of the conversion rule in **Term Builder**. Here is the rule (leq). The purpose of this rule is to draw the conclusion of observable equivalence, just like Leibniz Equality is meant to do.

$$\frac{\Gamma \vdash M, N : A \quad \Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash e : (M \approx_A N) \quad \Gamma \vdash P : A \to \mathsf{Type}}{\Gamma \vdash \mathsf{LeibnizEq}(M, N, e, P)\ :\ (P\ M) \to (P\ N)}\ (\mathsf{leq})$$

16 Soundness

16.1 Proof Overview

The approach to proving soundness of **Term Builder** is explained below. This, of course, would only be a “relative soundness”. The relativity comes from the fact that **Term Builder** does not check for well foundedness of types or non termination of functions, so it is up to the user not to allow these things (i.e. when the context file for a proof session is being prepared). Also, this applies to the rule of typing recursive functions. Even though it is a built-in rule, it is the only one that cannot be trusted unconditionally. The reason for such mistrust is that the rule (fix) can be used to program nonterminating terms, which leads to unsoundness in proofs. The approach that we take is to embed **Term Builder** without the rule (fix) within **CIC**, since soundness of **CIC** is a well known fact (reference [49]).

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{}{\Gamma \vdash \mathsf{Type} : \Box}$$

$$\frac{\Gamma \vdash A : \mathcal{D} \quad \mathcal{D} \in \{\mathsf{Type}, \Box\}}{\Gamma \vdash (\approx_A) : A \to A \to \mathsf{Type}} \qquad \frac{\Gamma, f : A \vdash M : A}{\Gamma \vdash (\mathit{fix}\ f : A.\ M) : A}\ (\mathsf{fix})$$

$$\frac{\Gamma \vdash A : \mathcal{D} \quad \Gamma, x : A \vdash B : \mathsf{Type}}{\Gamma \vdash (\forall x : A.B)\ :\ \mathsf{Type} \quad (\mathsf{all})_1} \qquad \frac{\Gamma \vdash A : \mathcal{D} \quad \Gamma, x : A \vdash B : \Box}{\Gamma \vdash (\forall x : A.B)\ :\ \Box \quad (\mathsf{all})_2}$$

$$\frac{\Gamma,\ x : A \vdash M : B \quad \Gamma \vdash (\forall x : A.B) : \mathcal{D}}{\Gamma \vdash (\lambda x : A.M) : (\forall x : A.B) \quad (\mathsf{abs})} \qquad \frac{\Gamma \vdash M : (\forall x : A.B) \quad \Gamma \vdash N : A}{\Gamma \vdash (M\ N) : B\{N/x\} \quad (\mathsf{app})}$$

where $\mathcal{D} \in \{\mathsf{Type}, \Box\}$ and $(\forall x : A.B) \equiv (A \to B)$, if $x \notin FV(B)$

$$\frac{\Gamma \vdash t : C \quad \Gamma \vdash P : C \to \mathsf{Type} \quad \{\Gamma,\ \{x_{i,j} : X_{i,j}\}_{j=1}^{k_i} \vdash M_i : (P\ (C_i \mathbf{x_i}))\}_{i=1}^{n}}{\Gamma \vdash (\langle P \rangle\ \mathbf{case}\ (t)\ of\ (C_1 \mathbf{x_1}) \mapsto M_1 \mid \ldots \mid\ of\ (C_n \mathbf{x_n}) \mapsto M_n : (P\ t) \quad (\mathsf{case})}$$

where $C ::= \{C_1\ X_{1,1} \ldots X_{1,k_1} \mid \ldots \mid C_n\ X_{n,1} \ldots X_{n,k_n}\}$;

$$\frac{\Gamma,\ x : A \vdash M : B \qquad \Gamma \vdash N : A}{\Gamma \vdash Beta(\lambda x : A.M, N) : ((\lambda x : A.M)N \approx M\{N/x\})}\ (\mathsf{beta})$$

$$\frac{\Gamma \vdash (\langle P \rangle\ \mathbf{case}\ (t)\ \{of\ (C_i \mathbf{x_i}) \ \mapsto M_i(\mathbf{x_i})\}_{i=1}^{n}) : (P\ t)}{\Gamma \vdash Eval(CASE_j) : (CASE_j \approx_{P(C_j \mathbf{v_j})} M_j\{\mathbf{v_j}/\mathbf{x_j}\})}\ (\mathsf{eval})$$

where $CASE_j \equiv \langle P \rangle\ \mathbf{case}\ (C_j \mathbf{v_j})\ \{of\ (C_i \mathbf{x_i}) \mapsto M_i(\mathbf{x_i})\}_{i=1}^{n}$

$$\frac{\Gamma \vdash M, N : A : \mathcal{D} \in \{\mathsf{Type}, \Box\} \quad \Gamma \vdash e : (M \approx_A N) \quad \Gamma \vdash P : A \to \mathsf{Type}}{\Gamma \vdash \mathsf{LeibnizEq}(M, N, e, P)\ :\ (P\ M) \to (P\ N) \qquad (\mathsf{leq})}$$

Figure 14: Complete Set of Proof Rules of **Term Builder**

16.2 Translations between Term Builder and CIC

Definition 1. TB $\triangleright\ \Gamma \vdash t : A$ *means that term t has type A in context* Γ*, and that this is provable within* **Term Builder***.*

Definition 2. CIC $\triangleright\ \Gamma' \vdash A'$ *means that statement* A' *holds in context* Γ' *within the calculus of inductive constructions. Such a statement may be a type assignment, or beta-reducibility, or other meta property.*

(Qualifiers **TB** and **CIC** are optional, when clear from context).

Contexts	Interpretation
$[\![\emptyset]\!]$	$\emptyset$
$[\![\Gamma, x : A]\!]$	$[\![\Gamma]\!], x : [\![A]\!]$
Expression	Interpretation
$[\![x]\!]$	x
$[\![\mathsf{Type}]\!]$	Type'
$[\![\Box]\!]$	$\Box'$
$[\![\lambda x : A.\ M]\!]$	$\lambda x : [\![A]\!].[\![M]\!]$
$[\![\forall x : A.\ B]\!]$	$\forall x : [\![A]\!].[\![B]\!]$
$[\![(M\ N)]\!]$	$([\![M]\!]\ [\![N]\!])$
$[\![(C_i \mathbf{v_i})]\!]$	$(C_i [\![\mathbf{v_i}]\!])$ (C_i is a constructor)
$[\![\approx_T]\!]$	$\lambda x : [\![T]\!].\ \lambda y : [\![T]\!].\ \forall P : [\![T]\!] \rightarrow \mathsf{Type}'.\ (Px) \rightarrow (Py)$
$[\![Beta(\lambda x.M^B, N)]\!]$	$\lambda P : [\![B]\!]\{[\![N]\!]/x\} \rightarrow \mathsf{Type}'.\ \lambda p : (P[\![(\lambda x.M)N]\!]).\ p$
$[\![Eval(c^T)]\!]$	$\lambda P : [\![T]\!] \rightarrow \mathsf{Type}'.\ \lambda p : (P[\![c]\!]).\ p$
$[\![\mathsf{LeibnizEq}(M, N, e, P)]\!]$	$\lambda p : [\![(PM)]\!].\ p$
$[\![\langle P\rangle$ **case** $(t)^C$ $\{of\ (C_i \mathbf{x_i}) \mapsto M_i(\mathbf{x_i})\}_{i=1}^n]\!]$	$\mathsf{match}\ [\![t]\!]\ \mathsf{in}\ [\![C]\!]\ \mathsf{return}\ [\![P]\!]$ $\mathsf{with}\ \{(C_i \mathbf{x_i}) \mapsto [\![M_i(\mathbf{x_i})]\!]\}_{i=1}^n$

Table 7: Encoding of **Term Builder** in **CIC**

We introduce the operation $[\![\ ]\!]$, which represents the translation of **Term Builder** expressions into the calculus of inductive constructions. Table 16.2 represents the action of this operation.

16.2.1 Reflection of Term Builder in CIC

Theorem 16.1. *(Substitution Lemma) The following substitution property exists between* **Term Builder** *and* **CIC**. *If $M(x)$ and N are terms in* **Term Builder** *then the following syntactic identity holds:*

$$[\![M\{N/x\}]\!] \equiv [\![M]\!]\{[\![N]\!]/x\}$$

Proof. This lemma follows immediately by standard structural induction on M. $\Box$

Theorem 16.2. *If in* **Term Builder** $\Gamma \vdash t : (M \approx N)$, *for some t, then*

$$\textbf{CIC} \rhd [\![\Gamma]\!] \vdash ([\![M]\!] \longleftrightarrow_{\beta\iota} [\![N]\!])$$

Proof. Equality types in **Term Builder** can originate from β-reduction, or from **case** statement evaluation. We consider these two cases below:

• Suppose that the **Term Builder** derivation ends with this clause:

$$\mathbf{TB} \ \triangleright \ \Gamma \vdash \mathit{Beta}(\lambda x^A.M, N) : (M\{N/x\} \approx (\lambda x^A.M)N)$$

The following line of reasoning gives us the desired result:

$$\begin{array}{ll}
(\lambda x : [\![A]\!].[\![M]\!])[\![N]\!] \longleftrightarrow_{\beta\iota} [\![M]\!]\{[\![N]\!]/x\} & \text{(by } \beta\text{-reduction in } \mathbf{CIC}\text{)} \\
(\lambda x : [\![A]\!].[\![M]\!])[\![N]\!] \longleftrightarrow_{\beta\iota} [\![M\{N/x\}]\!] & \text{(by substitution lemma)} \\
[\![(\lambda x : A.M)N]\!] \longleftrightarrow_{\beta\iota} [\![M\{N/x\}]\!] & \text{(by definition of the encoding)}
\end{array}$$

• Now let C be an algebraic type with constructors $\{C_i\}_{i=1}^n$, and let $t \equiv (C_j\mathbf{v_j})$. Suppose now that our **Term Builder** derivation ends with the following equality:

$$\mathbf{TB} \ \triangleright \ \Gamma \vdash \mathit{Eval}(CASE_j) : (CASE_j \approx M_j\{\mathbf{v_j}/\mathbf{x_j}\})$$

We need to validate the statement $\mathbf{CIC} \ \triangleright \ [\![CASE_j]\!] \longleftrightarrow_{\beta\iota} [\![M_j\{\mathbf{v_j}/\mathbf{x_j}\}]\!]$. First, by definition:

$$\begin{array}{rcl}
CASE_j & \equiv & \langle P\rangle \ \mathbf{case} \ (C_j\mathbf{v_j}) \ \{of \ (C_i\mathbf{x_i}) \mapsto M_i(\mathbf{x_i})\}_{i=1}^n \\
[\![CASE_j]\!] & \equiv & \mathsf{match} \ (C_j[\![\mathbf{v_j}]\!]) \ \mathsf{in} \ [\![C]\!] \ \mathsf{return} \ [\![P]\!] \ \mathsf{with} \ \{(C_i\mathbf{x_i}) \mapsto [\![M_i(\mathbf{x_i})]\!]\}_{i=1}^n
\end{array}$$

$$\begin{array}{rlll}
[\![CASE_j]\!] & \longrightarrow_{\beta\iota} & [\![M_j]\!]\{[\![\mathbf{v_j}]\!]/\mathbf{x_j}\} & \text{(by the reduction in } \mathbf{CIC}\text{)} \\
 & \equiv & [\![M_j\{\mathbf{v_j}/\mathbf{x_j}\}]\!] & \text{(by substitution lemma)}
\end{array}$$

□

Theorem 16.3. *If* **TB** $\triangleright \ \Gamma \vdash t : A$ *then* **CIC** $\triangleright \ [\![\Gamma]\!] \vdash [\![t]\!] : [\![A]\!]$.

Proof. We will prove the statement by structural induction over the type derivation. The base cases correspond to the axioms of the type systems, and the inductive cases are based on the remaining type rules.

• First let us look at what happens to the axioms of **CIC** and **Term Builder**. Consider the derivation step:

$$\frac{\text{(axiom)}}{\mathbf{TB} \ \triangleright \ \Gamma, x : A \vdash x : A}$$

We need to validate that $\mathbf{CIC} \rhd [\![\Gamma, x : A]\!] \vdash [\![x]\!] : [\![A]\!]$. By definition, $[\![\Gamma, x : A]\!] = [\![\Gamma]\!], x : [\![A]\!]$ and $[\![x]\!] = x$. Therefore, in **CIC** we have:

$$\begin{array}{l} \mathbf{CIC} \rhd [\![\Gamma]\!], x : [\![A]\!] \vdash x : [\![A]\!] \\ \mathbf{CIC} \rhd [\![\Gamma, x : A]\!] \vdash [\![x]\!] : [\![A]\!] \end{array}$$

Now consider the axiom $\Gamma \vdash \mathsf{Type} : \Box$. This is true for any context Γ, so we do not have to worry about the transformation into $[\![\Gamma]\!]$. We need to validate

$$\mathbf{CIC} \rhd [\![\Gamma]\!] \vdash [\![\mathsf{Type}]\!] : [\![\Box]\!].$$

By definition, $[\![\mathsf{Type}]\!] = \mathsf{Type}'$ and $[\![\Box]\!] = \Box'$. By the rule of **CIC**, we have:

$$\mathbf{CIC} \rhd [\![\Gamma]\!] \vdash \mathsf{Type}' : \Box'.$$

• Now let us look at the **Term Builder** derivation that ends with this step:

$$\frac{\mathbf{TB} \rhd \Gamma, x : A \vdash M : B \qquad \mathbf{TB} \rhd \Gamma \vdash (\forall x : A.B) : U \in \{\mathsf{Type}, \Box\}}{\mathbf{TB} \rhd \Gamma \vdash (\lambda x : A.M) : (\forall x : A.B)} \ (\mathsf{abs})$$

First, $[\![\Gamma, x : A]\!] = [\![\Gamma]\!], x : [\![A]\!]$. Then we have:

$$\begin{array}{ll} \mathbf{CIC} \rhd [\![\Gamma]\!] \vdash [\![\forall x : A.B]\!] : [\![U]\!] & \text{(by inductive hypothesis)} \\ \mathbf{CIC} \rhd [\![\Gamma]\!] \vdash \forall x : [\![A]\!].[\![B]\!] : [\![U]\!] & \text{(by definition of encoding)} \\ \mathbf{CIC} \rhd [\![\Gamma, x : A]\!] \vdash [\![M]\!] : [\![B]\!] & \text{(by inductive hypothesis)} \\ \mathbf{CIC} \rhd [\![\Gamma]\!], x : [\![A]\!] \vdash [\![M]\!] : [\![B]\!] & \text{(by definition of encoding)} \\ \mathbf{CIC} \rhd [\![\Gamma]\!] \vdash (\lambda x : [\![A]\!].[\![M]\!]) : (\forall x : [\![A]\!].[\![B]\!]) & \text{(by abstraction rule of } \mathbf{CIC}\text{)} \\ \mathbf{CIC} \rhd [\![\Gamma]\!] \vdash [\![(\lambda x : A.M)]\!] : [\![(\forall x : A.B)]\!] & \text{(by definition of encoding)} \end{array}$$

• Here we look at another case of our proof, the derivation ending in:

$$\frac{\mathbf{TB} \rhd \Gamma \vdash A : U_1 \qquad \mathbf{TB} \rhd \Gamma, x : A \vdash B : U_2 \qquad U_{1,2} \in \{\mathsf{Type}, \Box\}}{\mathbf{TB} \rhd \Gamma \vdash (\forall x : A.B) : U_2} \ (\mathsf{all})_{1,2}$$

We need to validate that $\mathbf{CIC} \rhd [\![\Gamma]\!] \vdash [\![(\forall x : A.B)]\!] : [\![U_2]\!]$. By inductive hypothesis, in **CIC** we have $\mathbf{CIC} \rhd [\![\Gamma]\!] \vdash [\![A]\!] : [\![U_1]\!]$ and also $\mathbf{CIC} \rhd [\![\Gamma, x : A]\!] \vdash [\![B]\!] : [\![U_2]\!]$. $[\![U_2]\!]$ can be either Type' or $\Box'$. In either case, we have:

$$\begin{array}{ll}
\mathbf{CIC} \ \triangleright\ [\![\Gamma, x : A]\!] \vdash [\![B]\!] : [\![U_2]\!] & \text{(by inductive hypothesis)}\\
\mathbf{CIC} \ \triangleright\ [\![\Gamma]\!], x : [\![A]\!] \vdash [\![B]\!] : [\![U_2]\!] & \text{(by definition of encoding)}\\
\mathbf{CIC} \ \triangleright\ [\![\Gamma]\!] \vdash \forall x : [\![A]\!].[\![B]\!] : [\![U_2]\!] & \text{(by rule (all) of }\mathbf{CIC}\text{)}\\
\mathbf{CIC} \ \triangleright\ [\![\Gamma]\!] \vdash [\![(\forall x : A.B)]\!] : [\![U_2]\!] & \text{(by definition of encoding)}
\end{array}$$

• Consider the derivation that ends with this step:

$$\frac{\mathbf{TB}\ \triangleright\ \Gamma \vdash M : (\forall x : A.B) \qquad \mathbf{TB}\ \triangleright\ \Gamma \vdash N : A}{\mathbf{TB}\ \triangleright\ \Gamma \vdash (M\ N) : B\{N/x\}}\ \textsf{(app)}$$

By inductive hypothesis, and by substitution lemma, we have:

$$\begin{array}{ll}
\mathbf{CIC} \ \triangleright\ [\![\Gamma]\!] \vdash [\![M]\!] : [\![\forall x : A.B]\!] & \text{(by inductive hypothesis)}\\
\mathbf{CIC} \ \triangleright\ [\![\Gamma]\!] \vdash [\![M]\!] : (\forall x : [\![A]\!].[\![B]\!]) & \text{(by definition of encoding)}\\
\mathbf{CIC} \ \triangleright\ [\![\Gamma]\!] \vdash [\![N]\!] : [\![A]\!] & \text{(by inductive hypothesis)}\\
\mathbf{CIC} \ \triangleright\ [\![\Gamma]\!] \vdash ([\![M]\!][\![N]\!]) : [\![B]\!]\{[\![N]\!]/x\} & \text{(by the rule (app) of }\mathbf{CIC}\text{)}\\
\mathbf{CIC} \ \triangleright\ [\![\Gamma]\!] \vdash [\![(M\ N)]\!] : [\![B\{N/x\}]\!] & \text{(by substitution lemma)}
\end{array}$$

• Let C be an algebraic type with constructors $\{C_i\}_{i=1}^n$. Suppose that our **Term Builder** derivation ends as follows: (we omit the qualifier **TB** $\triangleright$ here)

$$\frac{\Gamma \vdash t : C \quad \Gamma \vdash P : C \to \textsf{Type} \quad \{\Gamma,\ \{x_{i,j} : X_{i,j}\}_{j=1}^{k_i} \vdash M_i : (P\ (C_i\mathbf{x_i}))\}_{i=1}^n}{\Gamma \vdash (\langle P\rangle\ \mathbf{case}\ (t)\ \{of\ (C_i\mathbf{x_i}) \mapsto M_i\}_{i=1}^n : (P\ t) \quad \textsf{(case)}}$$

By inductive hypothesis $[\![\Gamma]\!] \vdash [\![t]\!] : [\![C]\!]$, $[\![\Gamma,\ \{x_{i,j} : X_{i,j}\}]\!]_{j=1}^{k_i} \vdash [\![M_i]\!] : [\![(P\ (C_i\mathbf{x_i}))]\!]$, and $[\![\Gamma]\!] \vdash [\![P]\!] : [\![C]\!] \to \textsf{Type}'$. Recall the equivalence:

$[\![\langle P\rangle\ \mathbf{case}\ (t)^C\ \{of\ (C_i\mathbf{x_i}) \mapsto M_i\}_{i=1}^n]\!] \qquad \equiv$

$\textsf{match}\ [\![t]\!]\ \textsf{in}\ [\![C]\!]\ \textsf{return}\ [\![P]\!]\ \textsf{with}\ \{(C_i\mathbf{x_i}) \mapsto [\![M_i]\!]\}_{i=1}^n$.

By rule (case) in **CIC**, the last expression has type $([\![P]\!][\![t]\!])$, which is equal to $[\![(P\ t)]\!]$. Therefore, we have:

$$\mathbf{CIC}\ \triangleright\ [\![\Gamma]\!] \vdash [\![\langle P\rangle\ \mathbf{case}\ (t)^C\ \{of\ (C_i\mathbf{x_i}) \mapsto M_i\}_{i=1}^n]\!] : [\![(P\ t)]\!]$$

• Suppose that the **Term Builder** derivation ends as follows:

$$\frac{}{\mathbf{TB}\ \triangleright\ \Gamma \vdash Beta(\lambda x.M, N) : ((\lambda x.M)N \approx_{\mathcal{D}} M\{N/x\})}\ \textsf{(beta)}$$

The following argument covers this case: let $\alpha \equiv (\lambda x.M)N$, and $\gamma \equiv M\{N/x\}$.

$$\begin{array}{ll}
\textbf{CIC} \,\triangleright\, [\![\Gamma]\!] \vdash ([\![\alpha]\!] \longleftrightarrow_{\beta\iota} [\![\gamma]\!]) & \text{(by theorem 16.2)} \\
\textbf{CIC} \,\triangleright\, [\![\Gamma]\!], P : [\![\mathcal{D}]\!] \to \mathsf{Type}' \vdash (P[\![\alpha]\!]) \longleftrightarrow_{\beta\iota} (P[\![\gamma]\!]) & \text{(since } [\![\alpha]\!] \longleftrightarrow_{\beta\iota} [\![\gamma]\!]) \\
\textbf{CIC} \,\triangleright\, [\![\Gamma]\!], P : [\![\mathcal{D}]\!] \to \mathsf{Type}',\ p : (P[\![\alpha]\!]) \vdash p : (P[\![\alpha]\!]) & \text{(by the axiom of } \textbf{CIC}) \\
\textbf{CIC} \,\triangleright\, [\![\Gamma]\!], P : [\![\mathcal{D}]\!] \to \mathsf{Type}',\ p : (P[\![\alpha]\!]) \vdash p : (P[\![\gamma]\!]) & \text{(by conversion rule in } \textbf{CIC})
\end{array}$$

Finally, by abstraction rule applied twice:

$\textbf{CIC} \,\triangleright\, [\![\Gamma]\!] \vdash (\lambda P : [\![\mathcal{D}]\!] \to \mathsf{Type}'.\ \lambda p : (P[\![\alpha]\!]).\ p) : (\forall P : [\![\mathcal{D}]\!] \to \mathsf{Type}'.\ (P[\![\alpha]\!]) \to (P[\![\gamma]\!]))$

This corresponds to: $\textbf{CIC} \,\triangleright\, [\![\Gamma]\!] \vdash [\![Beta(\lambda x.M, N)]\!] : [\![(\lambda x.M)N \approx_{\mathcal{D}} M\{N/x\}]\!]$

• Suppose that the **Term Builder** derivation ends as follows:

$$\frac{}{\textbf{TB} \,\triangleright\, \Gamma \vdash Eval(CASE_j) : (CASE_j \approx_T M_j\{\mathbf{v_j}/\mathbf{x_j}\})}\ (\mathsf{eval})$$

First, observe that:

$[\![Eval(CASE_j)]\!] \equiv \lambda P : [\![T]\!] \to \mathsf{Type}'.\ \lambda p : (P[\![CASE_j]\!]).\ p$

$[\![CASE_j \approx_T M_j\{\mathbf{v_j}/\mathbf{x_j}\}]\!] \equiv \forall P : [\![T]\!] \to \mathsf{Type}'.\ (P[\![CASE_j]\!]) \to (P[\![M_j\{\mathbf{v_j}/\mathbf{x_j}\}]\!])$

Now we continue to reason as follows: let $\alpha \equiv CASE_j$ and $\gamma \equiv M_j\{\mathbf{v_j}/\mathbf{x_j}\}$.

$$\begin{array}{ll}
\textbf{CIC} \,\triangleright\, [\![\Gamma]\!] \vdash ([\![\alpha]\!] \longleftrightarrow_{\beta\iota} [\![\gamma]\!]) & \text{(by theorem 16.2)} \\
\textbf{CIC} \,\triangleright\, [\![\Gamma]\!], P : [\![T]\!] \to \mathsf{Type}' \vdash (P[\![\alpha]\!]) \longleftrightarrow_{\beta\iota} (P[\![\gamma]\!]) & \text{(since } [\![\alpha]\!] \longleftrightarrow_{\beta\iota} [\![\gamma]\!]) \\
\textbf{CIC} \,\triangleright\, [\![\Gamma]\!],\ P : [\![T]\!] \to \mathsf{Type}',\ p : (P[\![\alpha]\!]) \vdash p : (P[\![\alpha]\!]) & \text{(by the axiom rule)} \\
\textbf{CIC} \,\triangleright\, [\![\Gamma]\!],\ P : [\![T]\!] \to \mathsf{Type}',\ p : (P[\![\alpha]\!]) \vdash p : (P[\![\gamma]\!]) & \text{(by the conversion rule)}
\end{array}$$

Finally, by the abstraction rule applied twice:

$$\textbf{CIC} \,\triangleright\, [\![\Gamma]\!] \vdash \lambda P : [\![T]\!] \to \mathsf{Type}'.\ \lambda p : (P[\![\alpha]\!]).\ p : \forall P : [\![T]\!] \to \mathsf{Type}'.\ (P[\![\alpha]\!]) \to (P[\![\gamma]\!])$$

Or, equivalently, $\textbf{CIC} \,\triangleright\, [\![\Gamma]\!] \vdash [\![Eval(CASE_j)]\!] : (CASE_j \approx_T M_j\{\mathbf{v_j}/\mathbf{x_j}\})$

• Suppose that the **Term Builder** derivation ends by typing the equality predicate:

$$\textbf{TB} \,\triangleright\, \Gamma \vdash\ (\approx_T)\ : T \to T \to \mathsf{Type}$$

We reason as follows:

$$\begin{array}{ll}
\textbf{CIC} \ \triangleright\ [\![\Gamma]\!], x : [\![T]\!], y : [\![T]\!], P : [\![T]\!] \to \mathsf{Type}' \vdash (Py) : \mathsf{Type}' & \text{(by rule } (\mathsf{app})) \\
\textbf{CIC} \ \triangleright\ [\![\Gamma]\!], x : [\![T]\!], y : [\![T]\!], P : [\![T]\!] \to \mathsf{Type}' \vdash (Px) : \mathsf{Type}' & \text{(by rule } (\mathsf{app})) \\
\textbf{CIC} \ \triangleright\ [\![\Gamma]\!], x : [\![T]\!], y : [\![T]\!], P : [\![T]\!] \to \mathsf{Type}' \vdash (Px) \to (Py) : \mathsf{Type}' & \text{(by rule } (\mathsf{all})_1) \\
\textbf{CIC} \ \triangleright\ [\![\Gamma]\!], x : [\![T]\!], y : [\![T]\!] \vdash \forall P : [\![T]\!] \to \mathsf{Type}'.\ (Px) \to (Py) : \mathsf{Type}' & \text{(by rule } (\mathsf{all})_1) \\
\textbf{CIC} \ \triangleright\ [\![\Gamma]\!] \vdash \lambda x.\ \lambda y.\ \forall P.\ (Px) \to (Py) : [\![T]\!] \to [\![T]\!] \to \mathsf{Type}' & \text{(by rule } (\mathsf{abs}))
\end{array}$$

By definition of the encoding, this is equivalent to:

$$\textbf{CIC} \ \triangleright\ [\![\Gamma]\!] \vdash [\![\approx_T]\!] : [\![T \to T \to \mathsf{Type}]\!]$$

• At last imagine that our **Term Builder** derivation ends as follows:

$$\frac{\textbf{TB} \ \triangleright\ \Gamma \vdash M, N : A \qquad \textbf{TB} \ \triangleright\ \Gamma \vdash e : (M \approx_A N) \qquad \ldots\ldots}{\textbf{TB} \ \triangleright\ \Gamma \vdash \mathsf{LeibnizEq}(M, N, e, P) : (P\ M) \to (P\ N)}\ (\mathsf{leq})$$

By Theorem 16.2 applied to term e: $\textbf{CIC} \ \triangleright\ [\![\Gamma]\!] \vdash ([\![M]\!] \longleftrightarrow_{\beta\iota} [\![N]\!])$.

And by inductive hypothesis $[\![M]\!], [\![N]\!] : [\![A]\!]$.

Since $[\![M]\!] \longleftrightarrow_{\beta\iota} [\![N]\!]$, we have: $([\![P]\!][\![M]\!]) \longleftrightarrow_{\beta\iota} ([\![P]\!][\![N]\!])$. By definition of our encoding, $[\![(PM)]\!] \longleftrightarrow_{\beta\iota} [\![(PN)]\!]$. Let $x : [\![(PM)]\!]$ in **CIC**. Then by rule (conv) and rule (abs),

$$\frac{\textbf{CIC} \ \triangleright\ [\![\Gamma]\!], x : [\![(PM)]\!] \vdash x : [\![(PN)]\!]}{\textbf{CIC} \ \triangleright\ [\![\Gamma]\!] \vdash (\lambda x : [\![(PM)]\!].\ x) : [\![(PM)]\!] \to [\![(PN)]\!]}\ (\mathsf{abs})$$

By definition of the encoding:

$$\textbf{CIC} \ \triangleright\ [\![\Gamma]\!] \vdash [\![\mathsf{LeibnizEq}(M, N, e, P)]\!] : [\![(PM) \to (PN)]\!]$$

□

16.2.2 Soundness Theorem

Theorem 16.4. Term Builder *is based on consistent logic, modulo rule* (fix).

Proof. Assume that **Term Builder** is inconsistent. Then $\textbf{TB} \triangleright \emptyset \vdash (\forall A : \mathsf{Type}.\ A)$. By Theorem 16.3, $\textbf{CIC} \ \triangleright\ \emptyset \vdash [\![(\forall A : \mathsf{Type}.\ A)]\!]$, or equivalently, $\textbf{CIC} \ \triangleright\ \emptyset \vdash (\forall A : \mathsf{Type}'.\ A)$ according to the encoding. Since the last statement is false, because **CIC** is consistent, **Term Builder** is also consistent. □

Part V

Using the Theorem Prover

17 User Interface of the Main Proof Window

The proof process starts by loading a context file, which contains all axioms, definitions, and type declarations. Thereafter, a formula to be shown is loaded from another file, and it must be compatible with the context, using only the notation predefined there. The prover itself works in two modes, *proof mode* and *use mode*. In *proof mode*, the so-called target formula on the bottom of the window is what we need to prove. In *use mode*, the formula on the bottom is already established, and we are looking to extract information from it to place into the context for later developments. The prover automatically detects what commands are appropriate in its current state. In section 18 we describe each such command, which is represented by a button on the main screen of **Term Builder**, what it does, and why it is useful.

The view of the main **Term Builder** window is presented in Table 8. The commands are situated on the right side, while the middle is reserved for the current context. By default we get the reflexivity proof constant and the identity function, whose type is equivalent with the constant `true`. Each proof term is tagged with the label, indicating the syntactic category to which the term belongs. The tags are as follows:

Table 8: Appearance of the Main Proof Window

Term Builder Context:	Add Axiom to Context ...	Mode: PROOF/USE
Term	*Type*	*Command*
[V] *Refl*	$\forall E$: Type. $(\forall e : E.\ e \approx_E e)$	Lambda abstract
[Λ] λA : Type...	$\forall A$: Type. $(A \rightarrow A)$	Generalize With...
		Add as Hypothesis
		Unify with Target
		Expand Definition
		Case Evaluation
		Enter Case Split...
		Upload Lemma...
		Enter Use Mode
		Instantiate With...
		Prove Antecedent
		Return from Use
		Use Reflexivity
		LHS ∩ LHS
		Match and Return
Proof	*Current Formula*	

[V]	variable
[λ]	lambda abstraction on the object level
[Λ]	lambda abstraction on the type level
[H]	inductive hypothesis
[A]	function application term
[C]	**case** expression
[Y]	recursive expression

In the next section we discuss each command on the main proof screen. Where applicable, we illustrate how the applications of these commands are translated into the internal proof rules. We show partial proof trees with the root on the bottom and the branches directed upwards, indicating the proof steps associated with corresponding prover commands.

18 Overview of Prover Commands

Load Axiom into Context. Axioms from the context file do not all appear in the visual context of the proof window. While some definitions of programmable functions and operators can be imported automatically by **Term Builder**, other axioms require explicit acquisition in order to be used by instantiation.

Lambda Abstract. Whenever the target formula has the shape $A \to B$ or $\forall x : A.\ B$, we have the option of detaching the logical assumption A into the context by issuing this command and continue to work on the proof of B, using assumption A. Here is the rule-based representation of this step, where the $\bullet$ characters indicate the places of the sought proof terms:

$$\boxed{\dfrac{\dfrac{\cdots}{\Gamma, x : A \vdash \bullet : B}}{\Gamma \vdash \bullet : (A \to B)} \ \text{by rule } (\mathsf{abs})}$$

Generalize With. If the current target formula has a free variable x, we might wish to strengthen it by placing the quantification $(\forall x)$ in front. One of the uses of such generalization in *proof mode* is to prepare to introduce a new induction hypothesis. Here is the proof step:

$$\boxed{\dfrac{\dfrac{}{\Gamma, x : A \vdash x : A} \qquad \dfrac{\cdots}{\Gamma \vdash \bullet : (\forall x : A.\ B(x))}}{\Gamma, x : A \vdash \bullet : B(x)} \ \text{by rule } (\mathsf{app})}$$

Add as Hypothesis. This command adds the current formula as an inductive hypothesis to the context. It may later be invoked into *use mode*, and instantiated. For an illustration, let us consider $\forall x : N.\ 0 + x \approx x$. This assertion happens to be inductive, so we go ahead and add it to the context as a hypothesis. We also immediately lambda abstract the bound variable x in the target, so that the context now looks as follows:

$$\Gamma, H : (\forall x : N.\ 0 + x \approx x),\ x : N \quad \vdash \quad 0 + x \approx x$$

We cannot use H immediately, we are only allowed to use it via induction on x. Namely, when we get to a point in our proof when a case split over the open variable x is performed, we will get a variable, which represents a substructure of what x represents. Then we can plug it into H, and proceed by induction, based on the definition of $(+)$ and other elements of the formula.

Unify with Target. For example, let our current target be $\alpha \approx \gamma$, and in the context we have $t : \alpha \approx \beta$. Suppose also that $\beta \approx \gamma$ is easier to prove than the target. Then we may want to unify t with the target to obtain an easier target $\beta \approx \gamma$. Below is a rule-based illustration of these manipulations:

$$\boxed{\dfrac{\Gamma \vdash t : (\alpha \approx \beta) \qquad \dfrac{\Gamma \vdash e : \beta \approx \gamma \quad \Gamma \vdash (\lambda x : X.\ \alpha \approx x) : (X \to \mathsf{Type})}{\Gamma \vdash \left\{\begin{array}{r}\mathsf{LeibnizEq}(\beta,\ \gamma,\ e,\ \lambda x : X.\ \alpha \approx x)\\ : (\alpha \approx \beta) \to (\alpha \approx \gamma)\end{array}\right\}}\ \text{by rule } (\mathsf{leq})}{\Gamma \vdash (\mathsf{LeibnizEq}(\beta, \gamma, e, \lambda x : X.\ \alpha \approx x)\ t) : (\alpha \approx \gamma)}\ \text{by rule } (\mathsf{app})}$$

Expand Definition. The user will be presented with a choice of functions or predicates, currently in the formula, in order to choose, which definition to unroll to proceed further. For example, if we had $(h\ k)$ as a choice for expansion, and our context had an axiom: $\mathit{def} : ((h) \approx \lambda x.H)$, then $(h\ k)$ would be replaced accordingly by an instantiation $H\{a/x\}$. The following is the rule-based view of the process:

Given the term E_1 of type $\mathcal{K}(h\ k)$, we want to perform an expansion of function symbol h, based on the definition def. Let us define the following abbreviations:

$$\begin{array}{lcll} L_1 & = & \mathsf{LeibnizEq}(h, (\lambda x.H), def, (\lambda f.Refl(f\ k))) & : (h\ k) \approx ((\lambda x.H)k) \\ B_1 & = & Beta((\lambda x.H)k, H\{k/x\}) & : ((\lambda x.H)k) \approx H\{k/x\} \\ L_2 & = & \mathsf{LeibnizEq}((h\ k), (\lambda x.H)k, L_1, \mathcal{K}) & : \mathcal{K}(h\ k) \rightarrow \mathcal{K}((\lambda x.H)k) \\ L_3 & = & \mathsf{LeibnizEq}((\lambda x.H)k, H\{k/x\}, B_1, \mathcal{K}) & : \mathcal{K}((\lambda x.H)k) \rightarrow \mathcal{K}(H\{k/x\}) \end{array}$$

It is now easy to present the final construction in the **Term Builder** type system:

$$\boxed{\dfrac{\dfrac{E_1 : \mathcal{K}(h\ k) \quad L_2 : \mathcal{K}(h\ k) \rightarrow \mathcal{K}((\lambda x.H)k)}{(L_2\ E_1) : \mathcal{K}((\lambda x.H)k)}\ \text{by rule } (\mathsf{app}) \qquad L_3 : \mathcal{K}((\lambda x.H)k) \rightarrow \mathcal{K}(H\{k/x\})}{(L_3(L_2\ E_1)) : \mathcal{K}(H\{k/x\})}\ \text{by rule } (\mathsf{app})}$$

Case Evaluation. This feature allows us to perform a desirable simplifying operation. When a **case** statement has an argument which matches a branch, it reduces to what is in that branch. As a target formula, let $\mathcal{K}$ be some context, and the **case** statement is inside. We will present what is happening inside the prover in terms of using proof rules with the following illustration:

Let $C ::= \{C_1\ X_{1,1}\ \ldots\ X_{1,k_1} \mid \ldots \mid C_n\ X_{n,1}\ \ldots\ X_{n,k_n}\}$ be an algebraic type, and

let $CASE_j \equiv (\langle P \rangle$ **case** $C_j \mathbf{v_j}\ \{of\ C_i \mathbf{x_i} \mapsto M_i\}_{i=1}^n)$. Then:

$$\boxed{\begin{array}{l} \left\{ \begin{array}{l} \Gamma \vdash t : C \\ \Gamma \vdash P : C \rightarrow \mathsf{Type} \end{array} \right\} \qquad \{\Gamma, \{x_{i,j} : X_{i,j}\}_{j=1}^{k_i} \vdash M_i : (P(C_i \mathbf{x_i}))\}_{i=1}^n \\ \hline \Gamma \vdash (\langle P \rangle\ \mathbf{case}\ t\ \{of\ C_i \mathbf{x_i} \mapsto M_i\}_{i=1}^n) : (P\ t) \quad \text{by rule } (\mathsf{case}) \\ \hline \Gamma \vdash Eval(CASE_j) : (CASE_j \approx M_j\{\mathbf{v_j}/\mathbf{x_j}\}) \quad \text{by rule } (\mathsf{eval}) \\ \Gamma \vdash \mathcal{K} : P(C_j \mathbf{v_j}) \rightarrow \mathsf{Type} \\ \hline \Gamma \vdash \left\{ \begin{array}{r} \mathsf{LeibnizEq}(M_j\{\mathbf{v_j}/\mathbf{x_j}\},\ CASE_j,\ Eval(CASE_j),\ \mathcal{K}) \\ : \mathcal{K}(M_j\{\mathbf{v_j}/\mathbf{x_j}\}) \rightarrow \mathcal{K}(CASE_j) \end{array} \right\} \quad \text{by rule } (\mathsf{leq}) \\ \Gamma \vdash k_0 : \mathcal{K}(M_j\{\mathbf{v_j}/\mathbf{x_j}\}) \\ \hline \Gamma \vdash \left\{ \begin{array}{r} (\mathsf{LeibnizEq}(M_j\{\mathbf{v_j}/\mathbf{x_j}\},\ CASE_j,\ Eval(CASE_j),\ \mathcal{K})\ k_0) \\ : \mathcal{K}(CASE_j) \end{array} \right\} \quad \text{by rule } (\mathsf{app}) \end{array}}$$

Enter Case Split. This is the least attractive and least economical feature (in terms of proof amount), but a necessary one for proofs by structural induction. Basically, upon selecting an open variable of an algebraic type A in current target formula, we split into cases based on the constructors of A. Suppose we work with Peano Arithmetic, and our split is based upon a variable $n : N$, where $N ::= \{0 \mid S\ N\}$, and we want to prove $\forall n : N.\ (P\ n)$. There will be two branches, one to prove $(P\ 0)$ and the other to prove $(P\ k)$ for $k > 0$. The types of the branches are $(P\ 0)$ and $(P\ k)$, but the type of the entire **case** operator is $(P\ n)$. We will illustrate the case split as follows. The proof terms $p : (P\ 0)$ and $q : (P(Sm))$ are exactly those that have to be completed in each of the two branches respectively:

$$\dfrac{\dfrac{\left\{\begin{array}{l}\Gamma, n : N \vdash n : N \\ \Gamma \vdash P : N \to \mathsf{Type}\end{array}\right\} \qquad \Gamma \vdash p : (P\ 0) \qquad \Gamma, m : N \vdash q : (P(Sm))}{\Gamma, n : N \vdash (\langle P \rangle\ \textbf{case}\ n\ of\ 0 \mapsto p \mid of\ Sm \mapsto q) : (P\ n)}\ \text{by rule } (\mathsf{case})}{\Gamma \vdash \lambda n : N.\ \langle P \rangle\ \textbf{case}\ n\ of\ 0 \mapsto p \mid of\ Sm \mapsto q) : (\forall n : N.\ (P\ n))}\ \text{by rule } (\mathsf{abs})$$

Upload Lemma. This command loads a proof prepared earlier into the **Term Builder** context, so it does type checking right after reading in a file, and the lemma may later become part of the current proof. We emphasize that what is loaded is a proof file in **Term Builder** language, not the statement of a lemma. The statement is inferred upon checking the proof.

Enter Use Mode. Normally we find ourselves in *proof mode*. However, we may have declared an induction hypothesis, or loaded an axiom, which we now want to use. What we do is we fetch it from the context and enter *use mode*. This mode is dual to *proof mode*. Statements like $\forall x : X.\ Y$ can be instantiated by a choice of expressions of type X. Also, if the current formula is $A \to B$, then A cannot be considered established in *use mode*, while in *proof mode* we must prove B as though A is already known to hold.

Instantiate With. This feature lets us utilize an established proposition when we are in *use mode*. For example, $\forall x : A.\ B$ can be instantiated with an available expression of type A. The rule-based view of this command is a simple application of Modus Ponens:

$$\boxed{\dfrac{\Gamma \vdash t : (\forall x : A.\ B) \qquad \Gamma \vdash a : A}{\Gamma \vdash (t\ a) : B\{a/x\}}\ (\mathsf{app})}$$

Prove Antecedent. Suppose we are in *use mode* and the formula that we want to use is $f : A \rightarrow B$. In order to use it, we have the option of proving A and then having B as an established fact. This command lets us do just that: we enter *proof mode* with the target A, and upon successfully returning back, we have $(a : A)$. It gets used automatically, and we end up with $(f\ a) : B$ as a formula for further use. The rule-based view of this command is again an application of Modus Ponens:

$$\boxed{\dfrac{\Gamma \vdash f : A \rightarrow B \qquad \Gamma \vdash a : A}{\Gamma \vdash (f\ a) : B}\ (\mathsf{app})}$$

Return From Use. Once we are done using a proposition for the time being, we exit the current *use mode*. After the exit, the current formula from the *use mode* screen gets moved to the context for future use.

Use REFL.

If our target formula has the form $K \approx K$, it gets proved via this feature.

LHS $\cap$ LHS. This feature is related to **Unify with Target**. Assume that our current target is $f(g(a)) \approx f(g(b))$. However, when f and g are uninterpreted, there is no immediate way to prove this even if $a \approx b$. Instead, if we add an equation $f(g(a)) \approx f(g(a))$ to the context, and then unify it with the target, the unification task will look like this:

$$t = \begin{cases} (Cons\ V_{126}\ V_{125}) \longrightarrow l = \begin{cases} (Cons\ V_{153}\ V_{152}) \\ null \longrightarrow V_{125} = \begin{cases} (Cons\ V_{560}\ V_{559}) \\ null \end{cases} \end{cases} \\ null \longrightarrow l = \begin{cases} (Cons\ V_{586}\ V_{585}) \\ null \end{cases} \end{cases}$$

Figure 15: Case Splitting Tree of the Proof

$$\begin{array}{|c|c|c|c|c|c|c|} f & g & a & \approx & f & g & a \\ f & g & a & \approx & f & g & b \end{array}$$

Each component on the top line would unify trivially with its counterpart on the bottom, except that the only remaining equality left to establish would be $a \approx b$. The further idea is explained by the following steps. Having constructed a proof $e : a \approx b$, we build the term $\mathsf{LeibnizEq}(a, b, e, \lambda x.f(g(a)) \approx f(g(x)))$. It has type $(f(g(a)) \approx f(g(a))) \rightarrow (f(g(a)) \approx f(g(b)))$. The antecedent is true by reflexivity, and we can conclude that our goal is reached by simple Modus Ponens.

Match and Return. This function lets us finish with the current screen, if we already have what is being asked. By selecting an item from the context that matches the goal, we complete the proof of the current target. For example, if we want to establish A and $\Gamma \vdash t : A$, then we can use the term t.

19 Example of a Proof Session

In this section we will demonstrate the functionality of **Term Builder** using an example combining the data type of lists, natural numbers, and arithmetic. The statement that we are going to prove is:

N	:	$\mathsf{Type}\ \{O \mid S\ N\};$
$NList$	:	$\mathsf{Type}\ \{null \mid Cons\ N\ NList\};$
$(+)$	:	$N \to N \to N;$
def_plus	:	$((+) \approx (\lambda x : N.\ \lambda y : N.\ case\ \langle \lambda n : N.\ N\rangle\ (y)$
		$\{of\ O \mapsto x\}$
		$\{of\ (Sw) \mapsto (S(x + w))\}));$
$append$	:	$NList \to NList \to NList;$
def_append	:	$(append \approx \lambda l : NList.\ \lambda t : NList.\ case\langle \lambda q : NList.\ NList\rangle\ (l)$
		$\{of\ null \mapsto t\}$
		$\{of\ (Cons\ n\ k) \mapsto (Cons\ n\ (append\ k\ t))\});$
len	:	$NList \to N;$
def_len	:	$(len \approx \lambda l : NList.\ case\langle \lambda q : NList.\ N\rangle\ (l)$
		$\{of\ null \mapsto O\}$
		$\{of\ (Cons\ n\ t) \mapsto (S(len\ t))\});$

Table 9: User Defined Context

$$\forall t : NList.\ \forall l : NList.\ (len(append\ t\ l)) \approx (len\ t) + (len\ l)$$

where N is the type of natural numbers, *NList* is the type of lists made of elements of N, *len* is the list length function, and *append* is the operation of list concatenation. All these components are user-defined. There is nothing that the system knows, even about the function (+), before the user provides necessary definitions. Once that is done, and only then, will this statement have the intended meaning that the length of the combined list is equal to the sum of two lengths. Table 9 shows the user defined context, necessary to carry out our proof. Each line is a declaration of a constant along with its type. As can be seen, some types are equality statements which serve to define functions. Figure 15 is the diagram of all the case splits which are done in this proof. Branching is based on the set of all constructors for a given data type. The equality sign following a variable indicates which expressions are used instead of it, after the branching point. The arrows indicate the flow of proof, i.e. which case choices lead to next case choices. We now proceed with the description of the proof session. We start the proof session by displaying the default screen with the formula to be proved at the bottom:

Term Builder	
Context:	Mode: PROOF
Term	*Type*
[V] *Refl*	$\forall E : \mathsf{Type}.\ (\forall e : E.\ e \approx_E e)$
$[\Lambda]\ \lambda A : \mathsf{Type}...$	$\forall A : \mathsf{Type}.\ (A \to A)$
Proof	*Current Formula*
	$\forall t : \mathit{NList}.\ \forall l : \mathit{NList}.\ (\mathit{len}(\mathit{append}\ t\ l)) \approx (\mathit{len}\ t) + (\mathit{len}\ l)$

At first, we should not forget to add the current formula as an inductive hypothesis to the context. It will appear there with the tag "H" as the type of a new automatically generated free variable named V_{86}.

Term Builder	
Context:	Mode: PROOF
Term	*Type*
[V] *Refl*	$\forall E : \mathsf{Type}.\ (\forall e : E.\ e \approx_E e)$
$[\Lambda]\ \lambda A : \mathsf{Type}...$	$\forall A : \mathsf{Type}.\ (A \to A)$
[H] V_{86}	$\forall t : \mathit{NList}.\ \forall l : \mathit{NList}.\ (\mathit{len}(\mathit{append}\ t\ l)) \approx (\mathit{len}\ t) + (\mathit{len}\ l)$
Proof	*Current Formula*
	$\forall t : \mathit{NList}.\ \forall l : \mathit{NList}.\ (\mathit{len}(\mathit{append}\ t\ l)) \approx (\mathit{len}\ t) + (\mathit{len}\ l)$

Recall that we restrict our use of induction to immediate sub-terms, therefore, term V_{86} could not be used here to "prove" the target formula immediately. Our induction principle works as follows: the base cases correspond to nullary constructors, (*null*, in this case), and the inductive hypothesis gets used in the branches belonging to the rest of constructors (*Cons*, in this case). Next, since we have two universal quantifiers in front of the target formula, we need to unbind the bound variables by pushing them into the context. This is done by using the lambda abstract command twice, and here is what we get:

Term Builder	
Context:	Mode: PROOF
Term	*Type*
[V] *Refl*	$\forall E : \mathsf{Type}.\ (\forall e : E.\ e \approx_E e)$
[Λ] $\lambda A : \mathsf{Type}...$	$\forall A : \mathsf{Type}.\ (A \rightarrow A)$
[H] V_{86}	$\forall t : NList.\ \forall l : NList.\ (len(append\ t\ l)) \approx (len\ t) + (len\ l)$
[V] t	$NList$
[V] l	$NList$
Proof	*Current Formula*
	$(len(append\ t\ l)) \approx (len\ t) + (len\ l)$

Now we are faced with the decision to expand the definition of one of the functions in our current formula. All these sub-expressions are listed in the choice window. Here we opt to expand the definition of equality. This equality is over type N, and is defined by the predicate of syntactic identity of internal structure:

$(n \approx m) = \textbf{case}\ m\ of\ O \mapsto (\textbf{case}\ n\ of\ O \mapsto \mathsf{true} \mid ofSk \mapsto \mathsf{false})$

$$\mid of\ Sw \mapsto (\textbf{case}\ n\ of\ O \mapsto \mathsf{false} \mid ofSk \mapsto (k \approx w));$$

Note that the tag "A" denotes that the grammatic category of the sub-expression is an application.

	Please make your Selection
[A]	$(len(append\ t\ l)) \approx (len\ t) + (len\ l)$
[A]	$(len(append\ t\ l))$
[A]	$(append\ t\ l)$
[A]	$(len\ t) + (len\ l)$
[A]	$(len\ t)$
[A]	$(len\ t)$

There are actually several unfoldings of function definitions that we do at this point. They are as follows:

Expand Definition of $\approx$ in:	$len(append\ t\ l) \approx (len\ t) + (len\ l)$
Expand Definition of *len* in:	$len(append\ t\ l)$
Expand Definition of *append* in:	$(append\ t\ l)$

After these unfoldings we enter the case splitter. We must split our reasoning based on two cases: is t an empty list, or not? This screen shows the splitter window. There are only two branches, since type *NList* has two constructors.

Case Splitter	
Argument :	t
Term	Branch
	$\{of\ null \mapsto l\}$
	$\{of\ Cons\ V_{126}\ V_{125} \mapsto (Cons\ V_{126}\ (append\ V_{125}\ l))\}$

We will work with the second branch first. As a result of unfolding function definitions in the previous screens, we have acquired unevaluated conditional statements from the bodies of the unfolded functions. After some trivial evaluation of these **case** statements we again enter the case splitter for the variable l. Since this variable is also of type *NList*, we get two branches:

Case Splitter	
Argument :	l
Term	Branch
	$\{of\ null \mapsto 0\}$
	$\{of\ Cons\ V_{153}\ V_{152} \mapsto (S\ (len\ V_{152})\}$

Here we also decided to work with the second branch first. After the two splits, the variables t and l have been decomposed, so that the formula that was our target before now looks like the current formula on the next screen:

Term Builder	
Context:	Mode: PROOF
Term	*Type*
[V] *Refl*	$\forall E : \mathsf{Type}.\ (\forall e : E.\ e \approx_E e)$
[Λ] $\lambda A : \mathsf{Type}$...	$\forall A : \mathsf{Type}.\ (A \to A)$
[H] V_{86}	$\forall t : NList.\ \forall l : NList.\ (len(append\ t\ l)) \approx (len\ t) + (len\ l)$
Proof	*Current Formula*
	$(len(append\ V_{125}\ (Cons\ V_{153}\ V_{152}))) \approx$ $(len\ (Cons\ V_{126}\ V_{125})) + (len\ V_{152})$

Evidently, $l = (Cons\ V_{153}\ V_{152})$ and $t = (Cons\ V_{126}\ V_{125})$. Now we want to use the induction hypothesis which is sitting in the context. It does not fully match the goal formula, but by proper instantiation it can get us closer to the desired form. So we enter the *use mode*, where V_{86} is the term whose type will be used:

Term Builder	
Context:	Mode: USE
Term	*Type*
[V] *Refl*	$\forall E : \mathsf{Type}.\ (\forall e : E.\ e \approx_E e)$
[Λ] $\lambda A : \mathsf{Type}\ldots$	$\forall A : \mathsf{Type}.\ (A \to A)$
[H] V_{86}	$\forall t : \mathit{NList}.\ \forall l : \mathit{NList}.\ (\mathit{len}(\mathit{append}\ t\ l)) \approx (\mathit{len}\ t) + (\mathit{len}\ l)$
Proof	*Current Formula*
[H] V_{86}	$\forall t : \mathit{NList}.\ \forall l : \mathit{NList}.\ (\mathit{len}(\mathit{append}\ t\ l)) \approx (\mathit{len}\ t) + (\mathit{len}\ l)$

The proper instantiation that we will use is plugging in V_{125} for t and leave l intact by plugging in $(\mathit{Cons}\ V_{153}\ V_{152})$ for l. Thereby we find ourselves in the following situation:

Term Builder	
Context:	Mode: USE
Term	*Type*
[V] *Refl*	$\forall E : \mathsf{Type}.\ (\forall e : E.\ e \approx_E e)$
[Λ] $\lambda A : \mathsf{Type}\ldots$	$\forall A : \mathsf{Type}.\ (A \to A)$
[H] V_{86}	$\forall t : \mathit{NList}.\ \forall l : \mathit{NList}.\ (\mathit{len}(\mathit{append}\ t\ l)) \approx (\mathit{len}\ t) + (\mathit{len}\ l)$
Proof	*Current Formula*
[A] $\ldots$	$(\mathit{len}(\mathit{append}\ V_{125}\ (\mathit{Cons}\ V_{153}\ V_{152}))) \approx (\mathit{len}\ V_{125}) + (\mathit{len}\ (\mathit{Cons}\ V_{153}\ V_{152}))$

Let us exit the *use mode* to get back to the *proof mode.* This will bring the current formula, which now possesses a proof, into the context as an established proposition.

Term Builder	
Context:	Mode: PROOF
Term	*Type*
[V] *Refl*	$\forall E : \mathsf{Type}.\ (\forall e : E.\ e \approx_E e)$
[Λ] $\lambda A : \mathsf{Type}...$	$\forall A : \mathsf{Type}.\ (A \rightarrow A)$
[H] V_{86}	$\forall t : NList.\ \forall l : NList.\ (len(append\ t\ l)) \approx (len\ t) + (len\ l)$
[A] ...	$(len(append\ V_{125}\ (Cons\ V_{153}\ V_{152}))) \approx$ $(len\ V_{125}) + (len\ (Cons\ V_{153}\ V_{152}))$
Proof	*Current Formula*
	$(len(append\ V_{125}\ (Cons\ V_{153}\ V_{152}))) \approx$ $(len\ (Cons\ V_{126}\ V_{125})) + (len\ V_{152})$

Now that we have obtained something that looks closer to the target formula, we take the direction of trying to unify one equality with another, where the left-hand-side should be unified by reflexivity.

Unifier	LHS	RHS
☐ *Refl*	$\approx$	$\approx$
☐ *Refl*	$len(append\ V_{125}\ (Cons\ V_{153}\ V_{152}))$	$len(append\ V_{125}\ (Cons\ V_{153}\ V_{152}))$
	$len(V_{125}) + (len(Cons\ V_{153}\ V_{152}))$	$len(Cons\ V_{126}\ V_{125}) + len(V_{152})$

By easy evaluation commands, the following two reductions are performed:

$$len(Cons\ V_{153}\ V_{152}) \longrightarrow S(len(V_{152}))$$

$$len(Cons\ V_{126}\ V_{125}) \longrightarrow S(len(V_{125}))$$

That is why the proof obligation to complete the requirements of the unifier looks as follows on the next main screen:

Term Builder	
Context:	Mode: PROOF
Term	*Type*
[V] *Refl*	$\forall E : \mathsf{Type}.\ (\forall e : E.\ e \approx_E e)$
[Λ] $\lambda A : \mathsf{Type}...$	$\forall A : \mathsf{Type}.\ (A \rightarrow A)$
[H] V_{86}	$\forall t : NList.\ \forall l : NList.\ (len(append\ t\ l)) \approx (len\ t) + (len\ l)$
[A] ...	$(len(append\ V_{125}\ (Cons\ V_{153}\ V_{152}))) \approx$ $(len\ V_{125}) + (len\ (Cons\ V_{153}\ V_{152}))$
Proof	*Current Formula*
	$(len\ V_{125}) + (S(len\ V_{152}))) \approx (S(len\ V_{125})) + (len\ V_{152})$

Instead of proving the current formula by induction from scratch, we will load the lemma, which proves just what we need. A lemma is something that we have proven in **Term Builder** before, and saved into a file. The lemma that we need here asserts that $\forall x : N.\forall y : N.\ x + Sy \approx Sx + y$. Here is the screen after we have loaded the lemma:

Term Builder	
Context:	Mode: PROOF
Term	*Type*
[V] *Refl*	$\forall E : \mathsf{Type}.\ (\forall e : E.\ e \approx_E e)$
[Λ] $\lambda A : \mathsf{Type}...$	$\forall A : \mathsf{Type}.\ (A \to A)$
[H] V_{86}	$\forall t : \mathit{NList}.\ \forall l : \mathit{NList}.\ (\mathit{len}(\mathit{append}\ t\ l)) \approx (\mathit{len}\ t) + (\mathit{len}\ l)$
[A] $\ldots$	$(\mathit{len}(\mathit{append}\ V_{125}\ (\mathit{Cons}\ V_{153}\ V_{152}))) \approx$ $(\mathit{len}\ V_{125}) + (\mathit{len}\ (\mathit{Cons}\ V_{153}\ V_{152}))$
[Y] $\mathit{Fix}\ldots$	$\forall x : N.\ \forall y : N.(x + Sy) \approx (Sx + y)$
Proof	*Current Formula*
	$(\mathit{len}\ V_{125}) + (S(\mathit{len}\ V_{152}))) \approx (S(\mathit{len}\ V_{125})) + (\mathit{len}\ V_{152})$

Now that the lemma is loaded, we immediately enter the *use mode* for this lemma:

Term Builder	
Context:	Mode: USE
Term	*Type*
[V] *Refl*	$\forall E : \mathsf{Type}.\ (\forall e : E.\ e \approx_E e)$
[Λ] $\lambda A : \mathsf{Type}...$	$\forall A : \mathsf{Type}.\ (A \to A)$
[H] V_{86}	$\forall t : \mathit{NList}.\ \forall l : \mathit{NList}.\ (\mathit{len}(\mathit{append}\ t\ l)) \approx (\mathit{len}\ t) + (\mathit{len}\ l)$
[A] $\ldots$	$(\mathit{len}(\mathit{append}\ V_{125}\ (\mathit{Cons}\ V_{153}\ V_{152}))) \approx$ $(\mathit{len}\ V_{125}) + (\mathit{len}\ (\mathit{Cons}\ V_{153}\ V_{152}))$
[Y] $\mathit{Fix}\ldots$	$\forall x : N.\ \forall y : N.(x + Sy) \approx (Sx + y)$
Proof	*Current Formula*
[Y] $\mathit{Fix}\ldots$	$\forall x : N.\ \forall y : N.(x + Sy) \approx (Sx + y)$

After the instantiation: $x \mapsto (\mathit{len}\ V_{125})$, $y \mapsto (\mathit{len}\ V_{152})$, we end up with the following established current formula on the main screen:

Term Builder	
Context:	Mode: USE
Term	*Type*
[V] *Refl*	$\forall E : \mathsf{Type}.\ (\forall e : E.\ e \approx_E e)$
[Λ] $\lambda A : \mathsf{Type}...$	$\forall A : \mathsf{Type}.\ (A \to A)$
[H] V_{86}	$\forall t : NList.\ \forall l : NList.\ (len(append\ t\ l)) \approx (len\ t) + (len\ l)$
[A] ...	$(len(append\ V_{125}\ (Cons\ V_{153}\ V_{152}))) \approx$ $(len\ V_{125}) + (len\ (Cons\ V_{153}\ V_{152}))$
[Y] $Fix\ldots$	$\forall x : N.\ \forall y : N.(x + Sy) \approx (Sx + y)$
Proof	*Current Formula*
[A] ...	$(len\ V_{125}) + S(len\ V_{125}) \approx S(len\ V_{125}) + (len\ V_{125})$

Now we are back to the unifier, where our proof obligation has been fulfilled, and we are ready to move on, after contemplating this complete picture:

Unifier	LHS	RHS
☐ *Refl*	$\approx$	$\approx$
☐ *Refl*	$len(append\ V_{125}\ (Cons\ V_{153}\ V_{152}))$	$len(append\ V_{125}\ (Cons\ V_{153}\ V_{152}))$
[A] ...	$len(V_{125}) + (len(Cons\ V_{153}\ V_{152}))$	$len(Cons\ V_{126}\ V_{125}) + len(V_{152})$

At this time the lower branch ($l \approx (Cons\ldots)$) is complete, and this is shown in the case splitter window below:

Case Splitter	
Argument :	l
Term	Branch
	$\{of\ null \mapsto 0\}$
[A] ...	$\{of\ Cons\ V_{153}\ V_{152} \mapsto (S\ (len\ V_{152})\}$

We turn to the other case, where ($l \approx null$). Based on this assumption, and using inductive hypothesis, we have made a substitution: $t \mapsto V_{125}, l \mapsto null$. After the simplifications, the target formula $len(append\ t\ l) \approx (len\ t) + (len\ l)$ becomes:

$$len(append\ V_{125}\ null) \approx (len\ V_{125})$$

Term Builder	
Context:	Mode: PROOF
Term	*Type*
[V] *Refl*	$\forall E : \mathsf{Type}.\ (\forall e : E.\ e \approx_E e)$
[Λ] λA : Type...	$\forall A : \mathsf{Type}.\ (A \rightarrow A)$
[H] V_{86}	$\forall t : \mathit{NList}.\ \forall l : \mathit{NList}.\ (\mathit{len}(\mathit{append}\ t\ l)) \approx (\mathit{len}\ t) + (\mathit{len}\ l)$
Proof	*Current Formula*
	$(\mathit{len}\ (\mathit{append}\ V_{125}\ \mathit{null})) \approx (\mathit{len}\ V_{125})$

This statement must also be proved by induction. In order to build an induction hypothesis we must generalize it by adding a quantifier $\forall V_{125}$ in front. This is because when we use the hypothesis, we must be able to instantiate it with an argument. Logically, this is not strengthening, but weakening, so this step is perfectly safe:

Term Builder	
Context:	Mode: PROOF
Term	*Type*
[V] *Refl*	$\forall E : \mathsf{Type}.\ (\forall e : E.\ e \approx_E e)$
[Λ] λA : Type...	$\forall A : \mathsf{Type}.\ (A \rightarrow A)$
[H] V_{86}	$\forall t : \mathit{NList}.\ \forall l : \mathit{NList}.\ (\mathit{len}(\mathit{append}\ t\ l)) \approx (\mathit{len}\ t) + (\mathit{len}\ l)$
Proof	*Current Formula*
	$\forall V_{125}.\ (\mathit{len}\ (\mathit{append}\ V_{125}\ \mathit{null})) \approx (\mathit{len}\ V_{125})$

Now that we have what we wanted as a hypothesis, we go ahead and add it into a context with the tag “H” using the “Add as Hypothesis” command:

Term Builder	
Context:	Mode: PROOF
Term	*Type*
[V] *Refl*	$\forall E : \mathsf{Type}.\ (\forall e : E.\ e \approx_E e)$
[Λ] λA : Type...	$\forall A : \mathsf{Type}.\ (A \rightarrow A)$
[H] V_{86}	$\forall t : \mathit{NList}.\ \forall l : \mathit{NList}.\ (\mathit{len}(\mathit{append}\ t\ l)) \approx (\mathit{len}\ t) + (\mathit{len}\ l)$
[H] V_{521}	$\forall V_{125}.\ (\mathit{len}\ (\mathit{append}\ V_{125}\ \mathit{null}) \approx (\mathit{len}\ V_{125})$
Proof	*Current Formula*
	$\forall V_{125}.\ (\mathit{len}\ (\mathit{append}\ V_{125}\ \mathit{null})) \approx (\mathit{len}\ V_{125})$

Having generalized and then pushed the inductive hypothesis into the context, we can immediately lambda-abstract again to get back to the original statement in order to proceed as usual. Only now, the abstracted variable over which we plan to use induction shall become part of the context:

Term Builder	
Context:	Mode: PROOF
Term	*Type*
[V] *Refl*	$\forall E : \text{Type}.\ (\forall e : E.\ e \approx_E e)$
[Λ] λA : Type...	$\forall A : \text{Type}.\ (A \to A)$
[H] V_{86}	$\forall t : NList.\ \forall l : NList.\ (len(append\ t\ l)) \approx (len\ t) + (len\ l)$
[H] V_{521}	$\forall V_{125}.\ (len\ (append\ V_{125}\ null) \approx (len\ V_{125})$
[V] V_{125}	$NList$
Proof	*Current Formula*
	$(len\ (append\ V_{125}\ null) \approx (len\ V_{125})$

To prove the current formula on the previous screen by induction, we need to case split, based on variable V_{125} being a *null* list or not. The case when $(V_{125} \approx null)$ is routine, therefore we show two consecutive splitter windows at once:

Case Splitter	
Argument :	V_{125}
Term	Branch
	$\{of\ null \mapsto null\}$
	$\{of\ Cons\ V_{560}\ V_{559} \mapsto (Cons(V_{560}\ (append(V_{559}\ null))\}$

Case Splitter	
Argument :	V_{125}
Term	Branch
[A] ...	$\{of\ null \mapsto null\}$
	$\{of\ Cons\ V_{560}\ V_{559} \mapsto (Cons(V_{560}\ (append(V_{559}\ null))\}$

After the substitution $V_{125} \mapsto (Cons\ V_{560}\ V_{559})$, a series of definition expansions and simplifying **case** evaluations will lead us to the following screen:

Term Builder	
Context:	Mode: USE
Term	*Type*
[V] *Refl*	$\forall E : \mathsf{Type}.\ (\forall e : E.\ e \approx_E e)$
[Λ] λA : Type...	$\forall A : \mathsf{Type}.\ (A \to A)$
[H] V_{86}	$\forall t : NList.\ \forall l : NList.\ (len(append\ t\ l)) \approx (len\ t) + (len\ l)$
[H] V_{521}	$\forall V_{125}.\ (len\ (append\ V_{125}\ null) \approx (len\ V_{125})$
Proof	*Current Formula*
	$(len\ (append\ V_{559}\ null) \approx (len\ V_{559})$

This is the exact place to use our induction hypothesis V_{521}, since the variable V_{559} is a product of decomposing V_{125}, and is technically a sub-term of it. So here we enter *use mode* with the hypothesis V_{521} and instantiate it immediately with V_{559}:

Term Builder	
Context:	Mode: USE
Term	*Type*
[V] *Refl*	$\forall E : \mathsf{Type}.\ (\forall e : E.\ e \approx_E e)$
[Λ] λA : Type...	$\forall A : \mathsf{Type}.\ (A \to A)$
[H] V_{86}	$\forall t : NList.\ \forall l : NList.\ (len(append\ t\ l)) \approx (len\ t) + (len\ l)$
[H] V_{521}	$\forall V_{125}.\ (len\ (append\ V_{125}\ null) \approx (len\ V_{125})$
Proof	*Current Formula*
[A] ...	$(len\ (append\ V_{559}\ null) \approx (len\ V_{559})$

At this point the case split on V_{125} is completed, and both branches have been provided with proofs, as shown below. Right after that, the automatic process consisting of "Match and Return" commands, completes the case split on l, and on the 2-nd branch of the case split on t. Therefore we show all three case-split screens sequentially, without more comment on them:

Case Splitter	
Argument :	V_{125}
Term	Branch
[A] ...	$\{of\ null \mapsto null\}$
[A] ...	$\{of\ Cons\ V_{560}\ V_{559} \mapsto (Cons(V_{560}\ (append(V_{559}\ null))\}$

Case Splitter	
Argument :	l
Term	Branch
[A] $(Eq\ldots$	$\{of\ null \mapsto 0\}$
[A] $(Eq\ldots$	$\{of\ Cons\ V_{153}\ V_{152} \mapsto (S\ (len\ V_{152})\}$

Case Splitter	
Argument :	t
Term	Branch
	$\{of\ null \mapsto l\}$
[A] ...	$\{of\ Cons\ V_{126}\ V_{125} \mapsto (Cons\ V_{126}\ (append\ V_{125}\ l))\}$

The remaining cases are when $(t \approx null)$. This leads to a necessity to case split on l again:

Case Splitter	
Argument :	l
Term	Branch
	$\{of\ null \mapsto 0\}$
	$\{of\ Cons\ V_{586}\ V_{585} \mapsto (S\ (len\ V_{585})\}$

The *null* branch is trivial and consists of several standard definition expansions and **case** evaluations. In the *Cons* branch, after a series of definition unfoldings and **case** evaluations, the target formula is reduced to the following form: $(len\ V_{585}) \approx (len\ null) + (len\ V_{585})$. We will derive it by using a previously proved lemma:

$$\forall l : NList.\ (len\ l) \approx (len\ null) + (len\ l)$$

We load this lemma, place it into *use mode* and instantiate it with V_{585}:

Term Builder	
Context:	Mode: USE
Term	*Type*
[V] *Refl*	$\forall E : \mathsf{Type}.\ (\forall e : E.\ e \approx_E e)$
[Λ] $\lambda A : \mathsf{Type}$...	$\forall A : \mathsf{Type}.\ (A \to A)$
[H] V_{86}	$\forall t : NList.\ \forall l : NList.\ (len(append\ t\ l)) \approx (len\ t) + (len\ l)$
[Y] $Fix\ldots$	$\forall l : NList.\ (len\ l) \approx (len\ null) + (len\ l)$
Proof	*Current Formula*
[A]...	$(len\ V_{585}) \approx (len\ null) + (len\ V_{585})$

We have come to the end of our proof session. After numerous "Match and Return" commands and successful exits from the case splitters, all remaining case splits on l and t will be completed, which is shown below on three separate screens, as well as the conclusion:

Case Splitter	
$Argument:$	l
Term	Branch
[A] $(Eq...$	$\{of\ null \mapsto 0\}$
	$\{of\ Cons\ V_{586}\ V_{585} \mapsto (S\ (len\ V_{585})\}$

Case Splitter	
$Argument:$	l
Term	Branch
[A] $(Eq...$	$\{of\ null \mapsto 0\}$
[A] $(Eq...$	$\{of\ Cons\ V_{586}\ V_{585} \mapsto (S\ (len\ V_{585})\}$

Case Splitter	
$Argument:$	t
Term	Branch
[C] $(case...$	$\{of\ null \mapsto l\}$
[A] $(Eq...$	$\{of\ Cons\ V_{126}\ V_{125} \mapsto (Cons\ V_{126}\ (append\ V_{125}\ l))\}$

Proof Completed:

Property: $\forall t : NList.\ \forall l : NList.\ len(append\ t\ l) \approx (len\ t) + (len\ l).$

Part VI

Concluding Remarks

In part II we have proposed a three-valued logic for use in applications which are most naturally modeled using partial functions. We have shown how the question of checking validity of formulas in this logic can be solved by checking the formula and its Type Correctness Condition (TCC). Both these checks can be done using standard two-valued semantics.

Future work includes using these ideas to develop a more general notion of validity in the presence of theories with sub-sorts and dealing with non-strict functions and predicates (those which, like the Boolean operators $\wedge$ and $\vee$ do not have the property that if one of their children evaluates to $\perp$, then the whole expression evaluates to $\perp$).

In part III we use partiality among other tools to build an abstract decision procedure for theories of algebraic data structures. Novel features of our treatment include the ability to handle mutually recursive types in many-sorted setting, a simpler presentation of the theory, an abstract declarative algorithm, and smarter splitting rules which can greatly enhance efficiency. Future work includes handling universally or existentially quantified formulas within this framework.

In part IV we have proposed a concept theorem prover to reason deductively about algebraic structures and functions operating on them. There are two contributions in part IV. First, we avoid the unwanted polymorphism as explained in subsection 15.4.1, which is due to the type-theoretic conversion rule. The second contribution is presentation. We have developed a GUI-based solution for interactive deduction, which has shown to be very convenient and effective.

Future work related to our theorem prover can include a multitude of directions. First, all logical connectors may be taken as primitives for user convenience, rather than their encoding in **System F**. Second, some of the forms of non-parametric polymorphism and function overloading would be very relevant features to add. One approach to this is given in reference [13].

Third, a different view of algebraic types may be employed and facilitate the use of dynamic dispatch. ADTs would only have one constructor each, and disjunction of them would simulate multiplicity of constructors. The behavior would then be split according to the type of the argument, like dynamic dispatch is meant to do, and the **case** statement would become obsolete.

Finally, one of the future goals is to generalize part IV to general inductive data types and understand how explicit equality and explicit **case** statements will be affected. The reason for complications is that we are not going to be dealing with free term algebras, which we have had with algebraic types.

References

[1] *The Coq Proof Assistant Reference Manual, version 8.0.* 2004.

[2] L. Bachmair, A. Tiwari, and L. Vigneron. Abstract congruence closure. *JAR*, 31:129–168, 2003.

[3] F. Barbanera and S. Berardi. "Proof-irrelevance out of Excluded-middle and Choice in the Calculus of Constructions". *Journal of Functional Programming*, 6(3):519–525, 1996.

[4] H. Barendregt. *Lambda Calculi with Types*, volume 2 of *Handbook of Logic in Computer Science.* 1992.

[5] B. Barras, P. Corbineau, B. Grégoire, H. Herbelin, and J. L. Sacchini. A new elimination rule for the calculus of inductive constructions. 5497:32–48, 2009.

[6] C. Barrett, I. Chikanian, and C. Tinelli. An abstract decision procedure for satisfiability in the theory of recursive data types. Technical Report TR2005-878, Department of Computer Science, New York University, Nov. 2005.

[7] C. Barrett, I. Chikanian, and C. Tinelli. An abstract decision procedure for satisfiability in the theory of recursive data types. In *Proceedings of PDPAR*, Aug. 2006.

[8] C. Barrett and C. Tinelli. "CVC3". In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07), Berlin, Germany*, Lecture Notes in Computer Science. Springer, 2007.

[9] P. Benacerraf and H. Putnam, editors. *Philosophy of Mathematics, 2nd Edition.* Cambridge University Press, 1988.

[10] S. Berezin, C. Barrett, I. Chikanian, M. Chechik, A. Gurfinkel, and D. L. Dill. A practical approach to partial functions in CVC Lite. In W. Ahrendt, P. Baumgartner, H. de Nivelle, S. Ranise, and C. Tinelli, editors, *Selected Papers from the Workshops on Disproving and the Second International Workshop on Pragmatics of Decision Procedures (PDPAR '04)*, volume 125(3) of *Electronic Notes in Theoretical Computer Science*, pages 13–23. Elsevier, July 2005. Cork, Ireland.

[11] M. P. Bonacina and M. Echenim. Generic theorem proving for decision procedures. Technical report, Università degli studi di Verona, 2006. Available at `http://profs.sci.univr.it/∼echenim/`.

[12] G. Bruns and P. Godefroid. "Model Checking Partial State Spaces with 3-Valued Temporal Logics". In *Proceedings of Proceedings of 11th International Conference on Computer-Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 274–287, Trento, Italy, 1999. Springer.

[13] G. Castagna, G. Ghelli, and G. Longo. *A Calculus for Overloaded Functions with Subtyping*. April 1992.

[14] T. Coquand. *An Analysis of Girard's Paradox*, volume 531 of *Rapports de Recherche, INRIA*. 1986.

[15] T. Coquand and C. Paulin-Mohring. "Inductively Defined Types". In *Proceedings of the International Conference on Computer Logic (COLOG-88)*, volume 417 of *LNCS*, 1988.

[16] P. de Groote, editor. *The Curry-Howard Isomorphism*, volume 8 of *Cahiers du Centre de Logique*. 1995.

[17] W. M. Farmer. A Partial Functions Version of Church's Simple Theory of Types. *The Journal of Symbolic Logic*, 55(3):1269–1291, 1990.

[18] H. Geuvers. *Calculus of Constructions and Higher Order Logic*, volume 8 of *Cahiers du Centre de Logique*, pages 139 – 191.

[19] H. Geuvers and F. Wiedijk. A Logical Framework with Explicit Conversions. In C. Schürmann, editor, *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages*, 2004.

[20] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. 1989.

[21] A. Gurfinkel and M. Chechik. "Multi-Valued Model-Checking via Classical Model-Checking". In *Proceedings of 14th International Conference on Concurrency Theory (CONCUR'03)*, volume 2761 of *LNCS*, September 2003.

[22] W. Hodges. *A Shorter Model Theory.* Cambridge University Press, 1997.

[23] M. Kerber and M. Kohlhase. A Mechanization of Strong Kleene Logic for Partial Functions. In A. Bundy, editor, *12th International Conference on Automated Deduction*, volume 814 of *LNAI*, pages 371–385. Springer Verlag, 1994.

[24] M. Kerber and M. Kohlhase. Mechanising Partiality without Re-Implementation. In *21st Annual German Conference on Artificial Intelligence*, volume 1303 of *LNAI*, pages 123–134. Springer Verlag, 1997.

[25] S. C. Kleene. *Introduction to Metamathematics.* New York: Van Nostrand, 1952.

[26] D. Kozen. Complexity of finitely presented algebras. In *Proceedings of the 9-th Annual ACM Symposium on Theory of Computing*, pages 164–177, 1977.

[27] V. Kuncak and M. Rinard. On the theory of structural subtyping. Technical Report MIT-LCS-TR-879, Massachusetts Institute of Technology, 2003.

[28] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types.* Wiley Teubner, 1996.

[29] F. Lucio-Carrasco and A. Gavilanes-Franco. A First Order Logic for Partial Functions. In *Proceedings STACS'89*, volume 349 of *LNCS*, pages 47–58. Springer, 1989.

[30] Z. Luo. "ECC, An Extended Calculus of Constructions". In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 385–395. IEEE Press, 1989.

[31] A. I. Mal'cev. On elementary theories of locally free universal algebras. *Soviet Mathematical Doklady*, 2(3):768–771, 1961.

[32] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[33] K. Meinke and J. V. Tucker. Universal algebra. In S. Abramsky, D. V. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1. Claredon Press, 1992.

[34] A. Miquel. *Calculus of Constructions with Universes.* Slides, Proofs-as-Programs Summer School. 2002.

[35] J. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[36] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the Association of Computing Machinery (JACM)*, 27(2):356–364, April 1980.

[37] D. C. Oppen. Reasoning about recursively defined data structures. *Journal of the Association of Computing Machinery (JACM)*, 27(3):403–411, July 1980.

[38] C. Paulin-Mohring. “Inductive Definitions in the System Coq - Rules and Properties”. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications (TLCA-93)*, volume 664 of *LNCS*, 1993.

[39] F. Pfenning and C. Paulin-Mohring. “Inductively Defined Types in the Calculus of Constructions”. In *Proceedings of the 5th International Conference on Mathematical Foundations of Programming Semantics*, pages 209–228. Springer-Verlag, 1990.

[40] E. Pimentel, B. Venneri, and J. Wells, editors. *Proceedings Fifth Workshop on Intersection Types and Related Systems*, volume 45 of *EPTCS*, 2010.

[41] T. Rybina and A. Voronkov. A decision procedure for term algebras with queues. *ACM Transactions on Computational Logic*, 2(2):155–181, Apr. 2001.

[42] N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial.* Computer Science Laboratory, SRI International, Menlo Park, CA, 1993. Also appears in Tutorial Notes, *Formal Methods Europe'93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.

[43] R. Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, 1984.

[44] V. Sjöberg and A. Stump. Equality, quasi-implicit products, and large eliminations. In *ITRS*, pages 90–100, 2010.

[45] M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and Foundations of Mathematics.* 2006.

[46] P. Tichy. Foundations of partial type theory. *Reports on Mathematical Logic*, 14:59–72, 1982.

[47] K. N. Venkataraman. Decidability of the purely existential fragment of the theory of term algebras. *JACM*, 34(2):492–510, Apr. 1987.

[48] B. Werner. "A Normalization Proof for an Impredicative Type System with Large Eliminations over Integers". In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, 1992.

[49] B. Werner. *Une Théorie des Constructions Inductives.* PhD thesis, Université Paris 7, 1994.

[50] T. Zhang, H. B. Sipma, and Z. Manna. Decision procedures for term algebras with integer constraints. In *Proceedings of IJCAR '04 LNCS 3097*, pages 152–167, 2004.

[51] T. Zhang, H. B. Sipma, and Z. Manna. Term algebras with length function and bounded quantifier alternation. In *Proceedings of TPHOLs*, 2004.