Notational Definition and Top-Down Refinement for Interactive Proof Development Systems

Timothy George Griffin

Ph. D. Thesis

TR 88-937
August 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

NOTATIONAL DEFINITION AND TOP-DOWN REFINEMENT FOR INTERACTIVE PROOF DEVELOPMENT SYSTEMS

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Timothy George Griffin
August 1988

NOTATIONAL DEFINITION AND TOP-DOWN REFINEMENT FOR INTERACTIVE PROOF DEVELOPMENT SYSTEMS

Timothy George Griffin, Ph.D.
Cornell University 1988

This thesis is concerned with issues related to the design and implementation of systems that support interactive proof development. The goal is to provide logic-independent frameworks that can be used in the design and implementation of component parts for such systems. In particular, this thesis will focus on two frameworks: a language-independent account of notational definition and a logic-independent account of a structure-oriented approach to the interactive construction of goal-directed proofs. These particular concerns were inspired and motivated by the Nuprl proof development system. However, the results presented in this thesis are meant to be applicable to a wide class of logics.

The formal account of notational definition begins with the presentation of a simply typed λ-calculus together with a method of representing expressions in a given language $\mathcal{L}$ as λ-expressions. The λ-calculus has a very simple abbreviative mechanism that allows a new constant, introduced by means of a δ-equation, to represent a closed expression. A new construct, called a Δ-equation, is introduced. It provides a higher-level means of introducing new notations that is close to the conventional style of definition. A Δ-equation is given meaning by a translation into a simple δ-equation. The implementation of mechanisms that support notational definitions in interactive proof development systems as well as applications to Nuprl are discussed.

The second part of the thesis presents a logic-independent framework for top-down proof development called *abstract refinement.* Rather than presenting, as is done for Nuprl, refinement and refinement rules as primitive notions that are extended by grafting on tactics, tactics are instead viewed as the basic notion from which refinement trees arise as tree-structured representations of the iterated composition of tactics. This is accomplished with the definition of a new construct called a *tactic tree* that extends the LCF tactics framework in a naturl way. Several examples of proof development using tactic trees are presented. One example demonstrates that the refinement rules and the extraction mechanism of Nuprl can be modeled in this framework.

Biographical Sketch

Timothy George Griffin was born on July 20, 1955, in Fort Atkinson, Wisconsin. He entered the University of Wisconsin at Madison in the fall of 1974 and received a B.S. in Mathematics in 1978. In the fall of 1983, he entered the graduate program in Computer Science at Cornell University. He received his M.S. in 1985 and completed the Ph.D. program in August of 1988. He has accepted a position at Rice University in Houston, Texas.

for my parents

Acknowledgements

I would like to thank my special committee members, Robert Constable, Tim Teitelbaum, and Anil Nerode, for their advice and encouragement. My adviser, Robert Constable, has been a great source of inspiration and support since he first welcomed me into the Prl group during my first year at Cornell. Working with Tim Teitelbaum has been very enjoyable and rewarding.

All those involved in the Nuprl and Synthesizer Generator projects deserve credit for providing a challenging and stimulating research environment. I want to express my special thanks to Bob Harper. In addition to introducing me to Nuprl, he has been a good friend. I am also grateful to Stewart Allen and Doug Howe for numerous discussions over the years and for their invaluable comments on drafts of this thesis. Many members of the Synthesizer Generator project provided invaluable guidance and advice without which the EFS could not have been implemented. Special thanks to Roger Hoover in this regard.

The LFCS group provided encouragement and support during my stay in Edinburgh. I give special thanks to Arnon Avron, Bob Harper, Furio Honsell, and Ian Mason for making my visit an enjoyable one.

I am indebted to Jim Hook and Gerral Pottinger of Odyssey Research Associates, Inc. for many stimulating discussions regarding notational definition.

Above all, I owe special gratitude to my good friends Micah Beck and Susan Mitchell. I truly cannot imagine having made it without them. I would also like to thank Micah for being instrumental in my choosing to venture into Computer Science over five years ago and for the continuous advice and encouragement that he has provided since, including of course, his helpful comments on drafts of this thesis.

Finalmente, eu gostaria de agradecer ao Carlos, Flávia, Luis, Xana, Camões (João), e a todos os meus amigos da Associação Luso-Brasileira pela sua simpatia e amizade. E, claro, gostaria de exprimir a minha imensa gratidão pelo amor e companhia da Irene, e pelos momentos inesqueciveis que passámos juntos nesta calma cidadezinha americana.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Goals of the thesis

This thesis is concerned with issues related to the design and implementation of systems that support interactive proof development. Such systems allow users to state assertions in some logic, to construct arguments in support of these assertions, and to store and organize results obtained. The goal is to provide logic-independent frameworks that can be used in the design and implementation of component parts for such systems. In particular, this thesis will focus on two frameworks: a language-independent account of notational definition and a logic-independent account of a structure-oriented approach to the interactive construction of goal-directed proofs.

These particular concerns were inspired and motivated by the Nuprl proof development system [Con86]. Nuprl introduced several design innovations that encourage a high-level style of machine-assisted proof development. Among these innovations are a syntactic component that implements terms and notational definitions and a structure-oriented editor, called a *refinement editor*, that supports the interactive construction of goal-directed proofs. Users can also write proof-valued programs and invoke them as inference rules. Together these components provide a new kind of environment for interactive proof development that has been extensively used to develop large examples of mathematics [Kre86,How86,Cle87,How87,How88,Bas88].

There are several motivations for providing logic-independent accounts of notational definition and refinement. First, although the innovations of Nuprl have been introduced in the context of supporting a particular type-theoretic logic, we believe that many of the system components can be successfully used in the implementation of proof development systems for a wide range of logics. A logic-independent approach makes these innovations more accessible to a wider audience.

Second, although the mechanism for notational definition in the current Nuprl system has proven to be indispensable, the mechanism is not well-behaved in several respects that will be discussed below. This thesis presents a solution to this problem that can be applied to a large class of languages.

Third, systems such as Nuprl are large and complex and, therefore, likely to contain errors. Rather than an *ad hoc* implementation of a logic, a general framework could provide a formal criterion with which to assess the correctness of, and enhance confidence in, an implementation.

Finally, a general framework could substantially reduce the effort involved in the task of implementing interactive proof development systems. This can be realized in two ways. A general framework can aid in identifying generic components that can be implemented once and used in different systems. In addition, it can aid in the design of tools that partially automate the implementation process.

Again, it should be emphasized that the results presented in this thesis are meant to be widely applicable and not limited to the task of redesigning and reimplementing Nuprl, although this is certainly one intended application. Nonetheless, to motivate this work, a brief overview of the Nuprl system is presented in the next section, after which the thesis is outlined.

1.2 A brief overview of Nuprl

Nuprl is an interactive proof development system designed and built at Cornell University under the direction of Robert Constable and Joseph Bates [Bat79,BC83,Con86]. It supports the interactive construction of proofs in a constructive type theory closely related to the type theories of Martin-Löf [ML71,ML73,ML,ML82]. See [Con86] for a complete description of the system.

At the highest level, a Nuprl user interacts with a library manager. A Nuprl library is a linearly ordered collection of objects The library manager maintains dependency links from declarations contained in objects to their uses contained in other objects occurring later in the library. Objects contain declarations of notational definitions, declarations of ML expressions, refinement trees, and bindings for the Nuprl evaluation mechanism. The library manager allows a user to select objects in a library for viewing, modification or deletion and automatically maintains dependency links between library objects. The system provides facilities for storing and retrieving libraries in and from the file system.

An important aspect of informal mathematical practice is the introduction and use of defined notations. Any system that intends to mechanize mathematical reasoning in a manner useful to working mathematicians must provide some mechanism supporting notational definitions, and Nuprl provides a very flexible facility for doing so. The declaration of such definitions has the form

```
DEF defname
```

```
new == term
```

where new is the new notation defined i. terms of term, which is a Nuprl term possibly containing instances of previously defined notations. The left-hand side new is a text string containing argument slots of the form <x:kind> where x is a variable. For example, the following is a simple definition.

```
DEF iff
<P:prop> <=> <Q:prop> == (<P> => <Q>) & (<Q> => <P>)
```

Such definitions are stored as objects in a library. These definitions define new templates that may be called up by name defname using a the structure-oriented editor for terms. The effect is to insert a template having "holes" where the argument slots appeared in new. The editor is then used to fill in these holes with Nuprl terms and variables. For example, invoking the definition of iff would insert a template

```
<prop> <=> <prop>
```

which them could be instantiated to the term

```
A <=> B
```

with any terms A and B. The designators kind an prop play no formal role and serves only to remind the user what kind of object is to be inserted at each slot in an instance of this notation.

The logic of Nuprl is a constructive type theory closely related to Martin-Löf's intuitionistic theory of types. To assert A is to assert that there is some object a that has type A. This is generalized with the judgment form

$$x_1 \in A_1, \ldots, x_n \in A_n \gg A$$

called a *sequent.* The A_i are called the *assumptions*, while A is called the *conclusion* of the sequent. Roughly speaking, such a sequent asserts that for all $a_1, \ldots, a_n$ with

$$a_1 \in A_1, a_2 \in A_2[a_1/x_1], \ldots, a_n \in A_n[a_1/x_n, \ldots, a_{n-1}/x_{n-1}]$$

there exists some a with $a \in A[a_1/x_n, \ldots, a_n/x_n]$. For a more precise and complete explanation of Nuprl logic and its semantics see Chapter 8 of [Con86] and Allen [All87a, All87b]

The logic of Nuprl has been called a refinement logic [CKB85,Con86] because of the goal-directed character of its deductive apparatus. Proofs in Nuprl are tree-structured objects, called *refinement trees*, with sequents and inference rules attached to nodes. The basic inference rule of Nuprl are presented in the top-down style as shown in Figure 1.2, where $\Gamma_0 \gg A_0$ is called the *goal*, R is the name of the rule with arguments

$$
\begin{array}{ll}
\Gamma_0 \gg A_0 & [a_0] \\
\text{by R with } e_1, \ldots, e_k & \\
\quad \Gamma_1 \gg A_1 & [a_1] \\
\quad \vdots & \vdots \\
\quad \Gamma_n \gg A_n & [a_n]
\end{array}
$$

Figure 1.1: A Nuprl refinement rule.

e_i, and the $\Gamma_i \gg A_i$ are *subgoals* produced by the rule. Nodes in refinement trees may also be *incomplete*, in which case they contain only a sequent. A user interacts with a structure-oriented editor RED (Refinement EDitor) to construct refinement trees. The most common operation is that of selecting an incomplete node and invoking an inference rule. The system is said to have the *constructible subgoal property* in that each rule produces the subgoals from the goal and its arguments (See Harper's thesis [Har85]).

Given a complete refinement tree with goal $\Gamma \gg A$ the system automatically constructs a term a that has type A under the assumptions Γ by a process called *extraction*. Each a_i, for $1 \leq i \leq n$, in the rule diagram above represents the term automatically constructed from a completed refinement tree of the i^{th} subgoal, and a_0 represents the term that is then constructed at this node. It should be noted that these terms, called *extracted terms*, are not displayed to the user during the construction of a refinement tree. They are presented in the rule diagram because they are important in documenting the relationship between extracted terms and refinement trees. If a user starts with a goal $\gg A$ that is viewed as a program specification and constructs a complete refinement tree, then the system automatically extracts a term a which can be viewed as a program that satisfies this specification. In this way programs can be synthesized from proofs of their specifications.

Constructing refinement trees with only the built-in rules of inference would quickly become very tedious. Nuprl extends the refinement apparatus by allowing users to define derive rules and proof-search procedures written in the programming language ML [CKB85]. ML was originally developed as the so-called metalanguage for the Edinburgh LCF [GMW79]. The original LCF system used ML to represent tactics and to construct derivations in a formal system called $PP\lambda$. The LCF approach has been applied to many other logics and plays a central role in systems such as Isabelle [Pau86], the Göteborg group's implementation of type theory [Pet82,NPS86].

In summary, the Nuprl system introduced several important innovations. First, Nuprl supports a structure-oriented proof editor that allows a user to construct proofs in a manner that is both easier and more natural than constructing proofs in a read-evaluate-print loop or in a stack-oriented fashion. This proof editor also encorporates the full power of tactics by allowing the user to invoke tactics as inference rules. Second, the Nuprl system provides a mechanism that allows a user to define new notations and to instantiate instances of defined notations in a structure-oriented manner. This mechanism works together with the ability to define derived rules of inference to provide the user with a powerful means of raising the the system's level of inference (see [How88]). Third, the Nuprl system offers a library mechanism that maintains dependencies between library objects and allows a user to select, view, and modify any object in a library and to create new objects at any library location.

1.3 An overview of the thesis

Nuprl's definition mechanism has proven to be indispensable in the construction of large libraries of mathematics. This mechanism is, however, deficient in several respects. Before discussing these deficiencies, a few words must be said about the implementation of defined notations in Nuprl. Each Nuprl term is tagged with display instructions that the system uses to display the term to a user. The definition mechanism simply allows users to define new display forms for terms on the right-hand side of definitional equations. Instances of defined notations in Nuprl are always fully expanded internally, with the user seeing only the defined display forms.

There are are several serious drawbacks to this approach. First, a user cannot be certain about the form of the fully expanded term because variables of the defined notation or of the fully expanded term can capture, or be captured by, hidden variables. For example, a user might think that some variable in a term is free only to discover that it is captured by a binding variable in the fully expanded term. Second, substitution may not be able to recover a defined display form when α-conversions take place on the binding variables of an instance of defined notation. As a result, substitution into instances of defined notations during proof construction can induce a cascade of unwanted definition expansions, much to the dismay of the user. Finally, reestablishing display forms after substitution is computationally expensive.

A general solution to this problem is presented that can be used for a large class of languages. Chapters 2–5 provide the formal account of notational definition[1]. Chapter 2 presents a typed λ-calculus together with a method of representing expressions in a given language $\mathcal{L}$ as λ-expressions. This is called Church's method of encoding languages because the basic idea first appeared in Church [Chu40]. The approach presented is, however, more general than Church's and is taken from the theory of expressions in the Edinburgh Logical Framework [AHM87,HHP87]. The λ-calculus has a

[1]Some of the material in Chapters 2–5 was previously presented in [Gri88a,Gri88b], and is presented here in an expanded and rewritten form.

very simple abbreviative mechanism that allows a new constant to represent a closed expression. These abbreviations are introduced with δ-equations of the form $d \stackrel{\delta}{=} e$ where d is a new constant and e is a closed λ-expression.

Chapter 3 begins with a closer look at the issues that must be addressed in the design and implementation of a well-behaved definition mechanism suitable for interactive proof development systems. It then builds on the λ-encoding of languages to develop a language-independent account of notational definition. A new type of equation, called a Δ-equation, is introduced as a high-level construct for the declaration of notational definitions that is closer to that used in conventional practice. A Δ-equation is given meaning by a translation into simple δ-equations.

Chapter 4 presents the definition of a function Exp^{Δ} that expands instances of defined notations while preserving most variable names. The chapter also includes a rather lengthy and tedious proof showing that this function is well-defined.

Chapter 5 is concerned with implementing mechanisms that support notational definitions in interactive proof development systems. A description is presented of the mechanism supporting notational abbreviations in the EFS [Gri87], which was implemented by the author with the Cornell Synthesizer Generator [RT84,RT85,RT88b, RT88a]. Application to the Nuprl system is also discussed.

In the second part of the thesis, a logic-independent account of refinement is presented in Chapters 6 and 7. Chapter 6 presents the basic definitions for an abstract account of refinement. At first glance, it might appear that refinement is intimately bound up with the design of the Nuprl logic, especially the notion of extraction. The development of *abstract refinement* demonstrates that this is not the case. The basic idea is actually quite simple: a logic-independent account of refinement is provided by turning the refinement model of Nuprl on its head. Rather than viewing refinement and refinement rules as primitive notions that are extended by grafting on tactics, tactics are viewed as the basic notion from which refinement trees arise as tree-structured objects that represented the iterated composition of tactics. This is accomplished with the definition of a new construct called a *tactic tree.* Instead of viewing the LCF tactic framework as conveniently extending the refinement paradigm, refinement is viewed as arising from a natural extension to the LCF tactics paradigm. The chapter concludes with a proposal for an implementation of a logic-independent component supporting tactic trees.

The tactic-tree approach has several advantages. First, it is a simpler account of refinement than the standard one presented for refinement in Nuprl. Second, the approach makes apparent the manner in which the Nuprl-style apparatus of refinement can be applied to a wide range of problem areas and logics. A tactic tree component could be implemented independently of a particular application. This component then could be instantiated to a particular logic simply by providing an appropriate collection of tactics. Third, the tactic-tree approach simplifies the task of explaining the relationship of a logic to its various refinement presentations, and in this way provides a clear conception of refinement by giving it a more direct proof-theoretic justification. This can increase confidence that an implementation of a logic is correct, and provide precise definition of the relationship of a refinement logic to conventional proof-theoretic concepts.

Chapter 7 presents several examples of proof development using tactic trees. The chapter begins with a discussion of formal systems and their representation with tactics. Two formal systems are presented together with tactics encoding them: Hoare's logic and a fragment of type theory. The type theory example shows how the refinement rules and the extraction mechanism of Nuprl can be modeled by tactic trees constructed over a collection of tactics based on this formal system. A hypothetical tactic tree implementation of type theory is contrasted with the refinement mechanism implemented in the current Nuprl system.

Finally, Chapter 8 concludes with a discussion of topics for future research.

Chapter 2

Language Representation

This chapter presents a simply typed λ-calculus together with a method of representing expressions in a given language $\mathcal{L}$ as λ-expressions. Section 2.1 defines the typed λ-calculus that has operators for pairing and projection and a simple abbreviation mechanism, called a δ-equation, that allows constants to abbreviate closed expressions. Most of the definitions in this section are fairly standard. The reader familiar with the simply typed λ-calculus should be aware of the non-standard definitions of Evars, Bvars, Vars, as well as the definitions of sets of well-formed expressions $\Lambda[\Sigma, \mathcal{D}, \mathcal{T}_1, \mathcal{T}_2]$ and sets of normal-form expressions $\mathcal{NF}^\tau[\Sigma, \mathcal{D}, \mathcal{T}_1, \mathcal{T}_2]$.

Section 2.2 outlines a Church-style [Chu40] method of using the typed λ-calculus for language representation.

2.1 A typed λ-calculus

This section presents the definition of a typed λ-calculus with pairing and a simple abbreviation mechanism that allows constants to represent closed terms.

2.1.1 Syntax.

First, the set of *types* will be defined. Given a nonempty finite set of *base types* T_0 define the set of $\mathcal{T}[T_0]$ of *types* over T_0 by the following grammar, where τ ranges over $\mathcal{T}[T_0]$ and t ranges over T_0:

$$\tau \ ::= \ t \ \mid \ \tau \rightarrow \tau \ \mid \ \tau \times \tau$$

The set $\mathcal{T}[T_0]$ is often written simply as $\mathcal{T}$ when T_0 is clear from context.

Assume that $\mathcal{V}$ is a denumerable set of *variables* with a denumerable subset $\mathcal{V}_\tau \subset \mathcal{V}$ for each $\tau \in \mathcal{T}$. There is a denumerable set of *constants* $\mathcal{C}$ with $\mathcal{V} \cap \mathcal{C} = \phi$. The set of *expressions* is the given by the following syntax, where τ ranges over $\mathcal{T}[T_0]$, x ranges over $\mathcal{V}$, and c ranges over $\mathcal{C}$:

$$e \ ::= \ x^\tau \ \mid \ c \ \mid \ \lambda x^\tau.e \ \mid \ e(e) \ \mid \ \langle e, e\rangle \ \mid \ \pi_1(e) \ \mid \ \pi_2(e)$$

The expression $\lambda x^\tau.e$ will often be written simply as $\lambda x.e$ when the type τ can be inferred from context.

Given an ordered list of variables $[x_1, \ldots, x_n]$, the following notations will often be used:

$$\begin{aligned}
\vec{x} &\equiv [x_1, x_2, \ldots x_n] \\
\lambda\vec{x}.e &\equiv \lambda x_1.\lambda x_2.\ldots.\lambda x_n.e \\
e(\vec{x}) &\equiv e(x_1)(x_2)\ldots(x_n) \\
\{\vec{x}\} &\equiv \{x_1, x_2, \ldots x_n\}
\end{aligned}$$

with the understanding that $\lambda[].e \equiv e$ and $e([]) \equiv e$.

A binary relation on expressions, $e_2 \subseteq e_1$, read e_2 is a subterm of e_1, is defined such that for any e_1, $e_1 \subseteq e_1$, and if $e_2 \subseteq e_1$, then for any e, $e_2 \subseteq e_1(e)$, $e_2 \subseteq \langle e_1, e\rangle$, $e_2 \subseteq \langle e, e_1\rangle$, $e_2 \subseteq \lambda x.e_1$, $e_2 \subseteq e(e_1)$, and $e_2 \subseteq \pi_i(e_1)$.

The free variables of an expression are defined in the usual way:

Definition 1 (Fvars) *For any expression e, define* Fvars[e] *to be the set of* free variables *of e.* Fvars[e] *is defined by induction on the structure of e:*

$$\begin{array}{ll}
\text{Fvars}[v] & \equiv \{v\} \\
\text{Fvars}[c] & \equiv \phi \\
\text{Fvars}[\lambda x.e_1] & \equiv \text{Fvars}[e_1] - \{x\} \\
\text{Fvars}[e_1(e_2)] & \equiv \text{Fvars}[e_1] \cup \text{Fvars}[e_2] \\
\text{Fvars}[\langle e_1, e_2\rangle] & \equiv \text{Fvars}[e_1] \cup \text{Fvars}[e_2] \\
\text{Fvars}[\pi_i(e_1)] & \equiv \text{Fvars}[e_1] \quad (i \in \{1, 2\})
\end{array}$$

An expression e is *closed* if Fvars[e] $= \phi$. The variable x in the expression $\lambda x.b$ is said to *bind* free occurrences of x in b, and that b and all subterms of b are in the *scope* of x.

The functions Bvars and Evars are defined to obtain the set of *binding variables* and the set of *expression variables* contained in a given expression.

Definition 2 (Bvars) *For any expression e, define* Bvars[e] *to be the set of* binding variables *of e.* Bvars[e] *is defined by induction on the structure of e:*

$$\begin{array}{ll}
\text{Bvars}[v] & \equiv \phi \\
\text{Bvars}[c] & \equiv \phi \\
\text{Bvars}[\lambda x.e_1] & \equiv \{x\} \cup \text{Bvars}[e_1] \\
\text{Bvars}[e_1(e_2)] & \equiv \text{Bvars}[e_1] \cup \text{Bvars}[e_2]
\end{array}$$

$$\begin{array}{ll}
\text{Bvars}[\langle e_1, e_2\rangle] & \equiv \text{Bvars}[e_1] \cup \text{Bvars}[e_2] \\
\text{Bvars}[\pi_i(e_1)] & \equiv \text{Bvars}[e_1] \quad \textit{(for } i \in \{1,2\}\textit{)}
\end{array}$$

Definition 3 (Evars) *For any expression e, define* Evars$[e]$ *to be the set of* expression variables *of e.* Evars$[e]$ *is defined by induction on the structure of e:*

$$\begin{array}{ll}
\text{Evars}[v] & \equiv \{v\} \\
\text{Evars}[c] & \equiv \phi \\
\text{Evars}[\lambda x.e_1] & \equiv \text{Evars}[e_1] \\
\text{Evars}[e_1(e_2)] & \equiv \text{Evars}[e_1] \cup \text{Evars}[e_2] \\
\text{Evars}[\langle e_1, e_2\rangle] & \equiv \text{Evars}[e_1] \cup \text{Evars}[e_2] \\
\text{Evars}[\pi_i(e_1)] & \equiv \text{Evars}[e_1] \quad \textit{(for } i \in \{1,2\}\textit{)}
\end{array}$$

Definition 4 (Vars) *For any expression e define the set*

$$\text{Vars}[e] \equiv \text{Evars}[e] \cup \text{Bvars}[e].$$

For example, if $e \equiv \lambda x.\lambda y.\lambda z.\langle x, w\rangle$ then

$$\begin{array}{lllllll}
\text{Fvars}[e] & \equiv & \{w\}, & & \text{Bvars}[e] & \equiv & \{x, y, z\}, \\
\text{Evars}[e] & \equiv & \{x, w\}, & & \text{Vars}[e] & \equiv & \{x, y, z, w\}.
\end{array}$$

The notation

$$e\left[\frac{e_1}{v_1} \cdots \frac{e_n}{v_n}\right].$$

is used to denote the standard notion of simultaneous capture-avoiding substitution into e of the e_i for free occurrences of the v_i (see for example Van Daalen [vD80]) and

$$e\left\{\frac{e_1}{v_1} \cdots \frac{e_n}{v_n}\right\}.$$

to denote the simultaneous *replacement* of free occurrences of v_i by e_i in e, that is, substitution without capture avoidance (see for example Huet [Hue75]).

Given two expressions e_1 and e_2, $e_1 =_\alpha e_2$ denotes the usual notion of α-equality, i.e. the equivalence relation on expressions generated by the renaming of bound variables.

Also, assume NewVar$[S]$ is a function that takes a finite set of variables S and returns a new variable $z \notin S$.

2.1.2 Typing expressions.

A *signature* Σ is a pair $(T_0, \mathcal{S})$ where T_0 is a non-empty finite set of base types and $\mathcal{S}$ is a finite set of pairwise distinct typed constants $\mathcal{S} = \{c_1^{\tau_1}, \ldots, c_k^{\tau_k}\}$ where $c_i \in \mathcal{C}$ and the $\tau_i \in \mathcal{T}[T_0]$.

Given $\Sigma = (T_0, S)$, define the relation $\mathcal{D} \vdash_\Sigma e \in \tau$ in Figure 2.1.2, (also written $\mathcal{D} \vdash e \in \tau$ or $\vdash e \in \tau$ when Σ or $\mathcal{D}$ are clear from the context), to indicate that the expression e has type τ. The meta-variable $\mathcal{D}$ will range over finite lists of δ-equations

$$\mathcal{D} \;=\; d_1 \stackrel{\delta}{=} e_1, \ldots, d_m \stackrel{\delta}{=} e_m$$

which allow for the binding of a constant to a closed expression. The d_i are new constants $d_i \in calC$ that do not occur in S. These δ-equations are not those of Barendregt [Bar84] but rather are a special case of the δ-equations of the AUTOMATH (Van Daalen [vD80]) languages. AUTOMATH δ-equations allow the constant d to be parameterized. This notion corresponds to a special case of the Δ-equation, as will be discussed further in the next chapter.

The list of δ-equations $\mathcal{D}$ is assumed to be *well-formed* in the sense that for each i,

$$d_i \notin \{d_1, \ldots, d_{i-1}\} \cup \mathcal{C},$$

d_i does not occur in e_k for $1 \leq k \leq i$, $\text{Fvars}[e_i] = \phi$ for each i, and that for each i there is some τ^i such that

$$d_1 \stackrel{\delta}{=} e_1, \ldots, d_{i-1} \stackrel{\delta}{=} e_{i-1} \vdash_\Sigma e_i \in \tau^i.$$

The empty list of δ-equations is written [], while $\mathcal{D}[d \stackrel{\delta}{=} e]$ denotes the list obtained by appending the equation $d \stackrel{\delta}{=} e$ to the end of $\mathcal{D}$.

The set of *well-formed* expressions is defined as follows:

Definition 5 ($\Lambda[\Sigma, \mathcal{D}, \mathcal{T}_1, \mathcal{T}_2]$) *Define $\Lambda[\Sigma, \mathcal{D}, \mathcal{T}_1, \mathcal{T}_2]$ as the set of well-typed expressions having type in $\mathcal{T}_1$ with all free variables having type in $\mathcal{T}_2$. That is, $e \in \Lambda[\Sigma, \mathcal{D}, \mathcal{T}_1; \mathcal{T}_2]$ iff*

1. $\mathcal{D} \vdash_\Sigma e \in \tau$ *and* $\tau \in \mathcal{T}_1$,
2. *if* $x^\tau \in \text{Fvars}[e]$ *then* $\tau \in \mathcal{T}_2$.

Define the set of all well-formed expressions, w.r.t. $\Sigma = (T_0, S)$ and $\mathcal{D}$, to be

$$\Lambda[\Sigma, \mathcal{D}] \equiv \Lambda[\Sigma, \mathcal{D}, \mathcal{T}[T_0], \mathcal{T}[T_0]].$$

This set will often be called simply the set of well-formed expressions and denoted as Λ when $\mathcal{D}$ and Σ are clear from the context.

2.1.3 Reduction and definitional equality.

A binary relation on expressions, called the one-step reduction relation, $\mathcal{R}^\mathcal{D}$, is defined as

$$\mathcal{R}^\mathcal{D} \;\equiv\; \beta \cup \eta \cup \pi_1 \cup \pi_2 \cup \delta^\mathcal{D}$$

$$\mathcal{D} \vdash_\Sigma x^\tau \in \tau$$

$$\mathcal{D} \vdash_\Sigma c \in \tau \quad (c^\tau \in \Sigma)$$

$$\frac{\mathcal{D} \vdash_\Sigma e \in \tau}{\mathcal{D} \vdash_\Sigma \lambda x^{\tau_1}.e \in \tau_1 \to \tau}$$

$$\frac{\mathcal{D} \vdash_\Sigma e_1 \in \tau_1 \to \tau_2 \quad \mathcal{D} \vdash_\Sigma e_2 \in \tau_1}{\mathcal{D} \vdash_\Sigma e_1(e_2) \in \tau_2}$$

$$\frac{\mathcal{D} \vdash_\Sigma e_1 \in \tau_1 \quad \mathcal{D} \vdash_\Sigma e_2 \in \tau_2}{\mathcal{D} \vdash_\Sigma \langle e_1, e_2 \rangle \in \tau_1 \times \tau_2}$$

$$\frac{\mathcal{D} \vdash_\Sigma e \in \tau_1 \times \tau_2}{\mathcal{D} \vdash_\Sigma \pi_i(e) \in \tau_i} \quad (i \in \{1, 2\})$$

$$\frac{\mathcal{D} \vdash_\Sigma e \in \tau \quad \text{Fvars}[e] = \phi}{\mathcal{D}, d \stackrel{\delta}{=} e, \mathcal{D}' \vdash_\Sigma d \in \tau}$$

Figure 2.1: The typing relation.

where

$$
\begin{aligned}
\beta &= \{((\lambda x.e)(e'), e[e'/x]) \mid x \in \mathcal{V}, e_1, e_2 \in \Lambda\}, \\
\eta &= \{(\lambda x.e(x), e) \mid x \in \mathcal{V}, e \in \Lambda, x \notin \text{Fvars}[e]\}, \\
\pi_1 &= \{(\pi_1(\langle e_1, e_2\rangle), e_1) \mid e_1, e_2 \in \Lambda\}, \\
\pi_2 &= \{(\pi_2(\langle e_1, e_2\rangle), e_2) \mid e_1, e_2 \in \Lambda\}, \\
\delta^{\mathcal{D}} &= \{(d, e) \mid d \in \mathcal{C}, e \in \Lambda, d \stackrel{\delta}{=} e \text{ in } \mathcal{D}\}.
\end{aligned}
$$

Often this is written as $\mathcal{R}$ instead of $\mathcal{R}^{\mathcal{D}}$ when $\mathcal{D}$ can be inferred from context. If r is a string composed of nonrepeating symbols from the set $\{\beta, \eta, \pi, \delta\}$, then an expression is in r-normal form, or is r-nf, if it does not contain a s-redex, where s is a symbol in r. The r-normal form of an expression e will be denoted as $\text{nf}^r[e]$. (The symbol π represents both π_1 and π_2.) An expression is in normal form if it is in $\beta\eta\pi\delta$-nf.

Define the relation $e_1 \longrightarrow_{\mathcal{R}} e_2$ (in Barendregt [Bar84] this is called the compatible closure of $\mathcal{R}$) as

1. $e_1 \longrightarrow_{\mathcal{R}} e_2$ if $(e_1, e_2) \in \mathcal{R}$,
2. if $e_1 \longrightarrow_{\mathcal{R}} e_2$ then $\pi_1(e_1) \longrightarrow_{\mathcal{R}} \pi_1(e_2)$, $\pi_2(e_1) \longrightarrow_{\mathcal{R}} \pi_2(e_2)$, and $\lambda x.e_1 \longrightarrow_{\mathcal{R}} \lambda x.e_2$,
3. if $e_1 \longrightarrow_{\mathcal{R}} e_2$ then for every expression e, $e_1(e) \longrightarrow_{\mathcal{R}} e_2(e)$, $e(e_1) \longrightarrow_{\mathcal{R}} e(e_2)$, $\langle e_1, e\rangle \longrightarrow_{\mathcal{R}} \langle e_2, e\rangle$, and $\langle e, e_1\rangle \longrightarrow_{\mathcal{R}} \langle e, e_2\rangle$.

This will often be written as $\longrightarrow$ when the subsrcipt can be inferred from context.

The relation $\longrightarrow$ is Church-Rosser and strongly normalizing for well-formed terms. This can be shown by a suitable mapping into a related λ-calculus, (one without δ-equations), such as the calculus of Pottinger [Pot78,Pot81], or Troelstra [Tro86], or Vrijer [dV87].

Define the set of *r-nf expressions* as follows:

Definition 6 ($\mathcal{NF}^r[\Sigma, \mathcal{D}, \mathcal{T}_1, \mathcal{T}_2]$) *Define $\mathcal{NF}^r[\Sigma, \mathcal{D}, \mathcal{T}_1, \mathcal{T}_2]$ as the set of well-typed r-nf expressions having type in $\mathcal{T}_1$ with all free variables having type in $\mathcal{T}_2$. That is, $e \in \mathcal{NF}^r[\Sigma, \mathcal{D}, \mathcal{T}_1, \mathcal{T}_2]$ iff*

1. *e is in r-nf,*
2. *$\mathcal{D} \vdash_\Sigma e \in \tau$ and $\tau \in \mathcal{T}_1$,*
3. *if $x^\tau \in \text{Fvars}[e]$ then $\tau \in \mathcal{T}_2$.*

Define the set of all normal form expressions, w.r.t. $\Sigma = (T_0, S)$ and $\mathcal{D}$, to be

$$\mathcal{NF}[\Sigma, \mathcal{D}] \;\equiv\; \mathcal{NF}^{\beta\eta\pi\delta}[\Sigma, \mathcal{D}, \mathcal{T}[T_0], \mathcal{T}[T_0]] \;\subset\; \Lambda[\Sigma, \mathcal{D}].$$

This set will often be called simply the set of normal form expressions and denoted as $\mathcal{NF}$ when $\mathcal{D}$ and Σ are clear from the context.

A binary relation on expressions $\twoheadrightarrow$ is defined as

1. $e_1 \twoheadrightarrow e_2$ if $e_1 =_\alpha e_2$,
2. $e_1 \twoheadrightarrow e_2$ if $e_1 \longrightarrow e_2$,
3. $e \twoheadrightarrow e$,
4. if $e_1 \twoheadrightarrow e_2$ and $e_2 \twoheadrightarrow e_3$ then $e_1 \twoheadrightarrow e_3$.

In words, $\twoheadrightarrow$ is the reflexive transitive closure of the relation obtained by forming the union of the α-equality relation and the $\longrightarrow$ relation. Note that expressions are not identified, as is usually done, with their equivalence class modulo α-equality. This is done to make it possible to keep track of the actual variable names used in an expression.

The symbol $=_\Lambda$ will denote the equivalence relation on expressions generated by the $\twoheadrightarrow$ relation. If $e_1 =_\Lambda e_2$ then e_1 and e_2 are said to be *definitionally equal.*

2.2 Representing languages in the λ-calculus à la Church

The treatment of language representation presented here is influenced most by the theory of expressions presented in the Edinburgh Logical Framework [HHP87,AHM87], from which the notions of a signature describing a language $\mathcal{L}$ and of the *adequacy* of an encoding have been adopted. This presentation has also been influenced by Martin-Löf's *system of arities*, in Nordström, Petersson and Smith [NPS86], and by Curry and Feys [CF58]. Similar approaches have been presented for representing both mathematical expressions (Andrews [And86], de Bruijn [dB70], and Hindley [HS86]) and programming language expressions (Huet and Lang [HL78] and Reynolds [Rey81]). An ambitious use of this approach can be found in the syntactic component of the Ergo Project [LPR+88], which uses a λ-encoding called higher-order abstract syntax [PE88]. All of these approaches have been influenced by the methods of Church [Chu40], and so λ-representation it is referred to here as the Church-style of encoding languages.

A language $\mathcal{L}$ is encoded in the λ-calculus by means of a signature $\Sigma^{\mathcal{L}} = (T_0^{\mathcal{L}}, S^{\mathcal{L}})$ so that each expression A of $\mathcal{L}$ can be represented as a λ-expression $\overline{A} \in \Lambda[\Sigma^{\mathcal{L}}]$. The set of base types $T_0^{\mathcal{L}}$ consists of one base type for each syntactic category of $\mathcal{L}$. Variables of $\mathcal{L}$ are represented by variables of the λ-calculus of the appropriate type. The set $S^{\mathcal{L}}$ consists of one constant c_{op} of type τ_{op} for each syntactic operator op of $\mathcal{L}$. By convention, and without loss of generality, the types of all syntactic constants are assumed to be of the form t or $\tau \to t$, where $t \in T_0^{\mathcal{L}}$. The type τ_{op} specifies the syntactic categories of the subterms and the binding structure of the operator corresponding to the constant c_{op}. The abstraction $\lambda x.b$ is used to represent all variable binding operations of $\mathcal{L}$. In this way the formal symbols and the formation rules of a language $\mathcal{L}$ are accounted for by typed constants and the type checking rules of the λ-calculus while notions of free and bound variables, substitution and definitional equality are inherited directly from those defined for the λ-calculus.

The set of expressions in the language $\mathcal{L}$ is then identified with the set of λ-expressions

$$\overline{\mathcal{L}} \equiv \{\overline{A} \mid A \text{ in } \mathcal{L}\}.$$

Such an encoding will, in most cases, allow for an *adequacy theorem* characterizing this set of λ-expressions as a set of normal form expressions

$$\overline{\mathcal{L}} = \bigcup_{t \in T_0} \mathcal{NF}^{\beta\eta\pi}[\Sigma, [], \{t\}, T_t]$$

for some collection of sets $\{T_t\}$ where for each $t \in T_0$, $T_t \subseteq T_0$.

For illustration, a signature $\Sigma^{fol} \equiv (T_0^{fol}, S^{fol})$ based on a small first order language with individuals, and propositions, and sets is developed. The grammar for this language is presented in Figure 2.2 together with typed constants to represent the syntactic operators, where $T_0^{fol} = \{\mathrm{i}, \mathrm{o}, \mathrm{set}\}$ represents the syntactic categories of individuals, propositions, and sets respectively. Given this encoding we have, for example,

$$\begin{aligned} \overline{A \wedge B} &\equiv \wedge(\langle \overline{A}, \overline{B} \rangle), \\ \overline{\forall x.A} &\equiv \forall(\lambda x^{\mathrm{i}}.\overline{A}), \end{aligned}$$

for propositions A and B.

It is not too difficult to check that

$$\overline{fol} = \mathcal{NF}^{\beta\eta\pi}[\Sigma^{fol}, [], \{\mathrm{i}, \mathrm{o}, \mathrm{set}\}, \{\mathrm{i}\}]$$

See [HHP87] for a discussion of the adequacy of this representation.

It is necessary to explain why the pairing operator has been included in the λ-calculus. A syntactic operator with n binding variables and k subexpressions can be viewed abstractly as

$$op(\vec{x}; N_1^{\vec{y_1}}, \ldots, N_k^{\vec{y_k}})$$

where $\vec{x} = \{x_1, \ldots, x_n\}$ are the binding variables and where the superscripted variables $\{\vec{y_i}\} \subseteq \{\vec{x}\}$ indicate which variables of $\vec{x}$ bind in the subexpression N_i. Such a syntactic

$$
\begin{array}{lcl}
i & ::= & x \\
 & | & 0 \\
 & | & i + i \\
 & | & i \times i \\
P & ::= & i = i \\
 & | & P \wedge P \\
 & | & P \vee P \\
 & | & P \Rightarrow P \\
 & | & \forall x.P \\
 & | & \exists x.P \\
 & | & i \in S \\
S & ::= & \phi \\
 & | & \mathcal{N} \\
 & | & \{x \in S \mid P\}
\end{array}
$$

c_{op}	τ_{op}
0	**i**
$+$	$(\mathbf{i} \times \mathbf{i}) \rightarrow \mathbf{i}$
$\times$	$(\mathbf{i} \times \mathbf{i}) \rightarrow \mathbf{i}$
$=$	$(\mathbf{i} \times \mathbf{i}) \rightarrow \mathbf{o}$
$\wedge$	$(\mathbf{o} \times \mathbf{o}) \rightarrow \mathbf{o}$
$\vee$	$(\mathbf{o} \times \mathbf{o}) \rightarrow \mathbf{o}$
$\Rightarrow$	$(\mathbf{o} \times \mathbf{o}) \rightarrow \mathbf{o}$
$\forall$	$(\mathbf{i} \rightarrow \mathbf{o}) \rightarrow \mathbf{o}$
$\exists$	$(\mathbf{i} \rightarrow \mathbf{o}) \rightarrow \mathbf{o}$
$\in$	$(\mathbf{i} \times \mathbf{set}) \rightarrow \mathbf{o}$
ϕ	**set**
$\mathcal{N}$	**set**
Ξ	$(\mathbf{set} \times (\mathbf{i} \rightarrow \mathbf{o})) \rightarrow \mathbf{set}$

Figure 2.2: Grammar and syntactic constants for a first-order language.

operator can be encoded in the typed λ-calculus with a constant c_{op} of the appropriate type as

$$c_{op}(\lambda \vec{y_1}.N_1) \cdots (\lambda \vec{y_k}.N_k),$$

For example, an operator

$$op(x, y; N_1, N_2^x, N_3^{x,y}, N_4^{x,y})$$

can be encoded as

$$c_{op}(N_1)(\lambda x.N_2)(\lambda x, y.N_3)(\lambda x, y.N_4).$$

However, by using the pairing operator this syntactic construct can be constructed without duplicating the variables x and y as

$$c'_{op}(\langle N_1, \lambda x.\langle N_2, \lambda y.\langle N_3, N_4\rangle\rangle\rangle).$$

Of course, this approach cannot always avoid the duplication of variables, for example an operator such as

$$op(x, y; N_1^x, N_2^y, N_3^{x,y})$$

cannot be represented without duplicating variables. Such situations seem to arise rarely and so do not seem to warrant abandoning the advantages of the λ-calculus as a framework for encoding languages.

Although the issue of variable duplication may not present itself in the encoding of the basic syntactic operators of a given language, such as the one presented in Figure 2.2, it does arise quickly when considering syntactic definitions. In the next chapter, a formal account of notational definition is built on this Church-style representation of languages. This approach thus inherits the generality of this representation and is able to describe notational definitions for a wide class of languages — exactly those languages that can be represented in the typed λ-calculus. These include the languages of first-order logic, higher-order logics, modal logics, and various type theories.

Chapter 3

Notational Definition

3.1 Motivation

In the course of developing a mathematical theory or proof it is a common practice to introduce new notation to represent notation that is previously understood. Indeed, the definition of abbreviative notation is so much a part of mathematical practice that it is difficult to imagine "doing mathematics" without it. This chapter presents a formal account that is intended to model the practice of introducing and using notational (abbreviative) definitions. It is doubtful that any one formalism could capture all of the various contrivances employed by mathematicians, most often informally, when fashioning such definitions. Still, the account presented here encompasses a large and useful class of abbreviative definitions.

The motivation behind this chapter and the next is a pragmatic one: to provide a framework useful in the design and implementation of secure proof system interfaces that accommodate, as much as possible, conventional mathematical practice. Of particular interest are interactive systems that support formal reasoning, allowing users to state mathematical assertions, to construct formal proofs of these assertions in some logic, and to organize results. Examples of such systems include AUTOMATH [dB70, dB80], the LCF project [GMW79,Pau85], the Interactive Proof Editor (IPE) [Bur86, Jon87], the Göteborg group's implementation of type theory [Pet82,NPS86], and the Nuprl project [Con86]. Of these systems, only Nuprl provides a mechanism supporting a style of notational definitions close to that used conventionally. However, as indicated in Chapter 1, certain deficiencies of the Nuprl definition mechanism were brought to light through a great deal of practical experience with the system. This experience in turn aided in the formulation of desirable properties for such a mechanism as well as pointing out the need for a general and formal solution.

First, the conventional practice of introducing and using notational definitions deserves closer scrutiny. Among other things, notational definitions serve to provide a

$$a \Leftrightarrow b \stackrel{\text{def}}{=} (a \Rightarrow b) \wedge (b \Rightarrow a) \tag{3.1}$$

$$a \mid b \stackrel{\text{def}}{=} \exists z.z > 0 \wedge a \times z = b \tag{3.2}$$

$$\forall x \in a.b^x \stackrel{\text{def}}{=} \forall x.(x \in a) \Rightarrow b \tag{3.3}$$

$$\forall x \in a, y \in b^x.c^{x,y} \stackrel{\text{def}}{=} \forall x \in a.\forall y \in b.c \tag{3.4}$$

$$\forall x, y \in a.b^{x,y} \stackrel{\text{def}}{=} \forall x \in a.\forall y \in a.b \tag{3.5}$$

$$\exists! x.a^x \stackrel{\text{def}}{=} \exists x.a \wedge (\forall y.a[y/x] \Rightarrow x = y) \tag{3.6}$$

Figure 3.1: Examples of notational definitions.

means of abbreviation, to isolate an interesting concept, or to generalize a concept by abstraction. Such distinctions will be ignored and the focus will be on the purely *syntactic* aspects of such definitions. The introduction and use of syntactic definitions are usually treated informally as nonessential adjuncts to a language — even when a mathematician is working in a formal system. The design and implementation of a secure definition facility requires that a formal treatment be given to the informal practice of introducing and using syntactic definitions. Both the logical nature of notational definitions and some special considerations arising from an interactive setting in which a user engages in a "dialogue with a proof system" [Mil87] are explored.

In general, assume that a formal description is given of the syntax of a language $\mathcal{L}$ that is to be extended with notational definitions. Figure 3.1 presents examples of notational definitions that might appear in a mathematics text, where $\mathcal{L}$ is a first-order language. (The following discussion is not meant to apply only to standard first-order languages; this language is used here only to provide a source of familiar examples.)

The abbreviative notations that will be considered, such as those of Figure 3.1, are introduced by means of *equations*. These equations have the form

$$lhs \stackrel{\text{def}}{=} rhs$$

where the left-hand side, *lhs*, denotes new a notation defined in terms of the right-hand side, *rhs*[1]. The *rhs* is an expression built out of expressions in $\mathcal{L}$, other instances of notational definitions, and possibly a substitution operator. Given expressions A

[1]We use the more mnemonic *lhs* and *rhs* rather than *definiendum* and *definiens*.

and B, the expression $A \Leftrightarrow B$ will be called an *instance* of the notation defined in Equation 3.1. This instance $A \Leftrightarrow B$ is understood to represent, or to stand for, the expression $(A \Rightarrow B) \wedge (B \Rightarrow A)$, which will be called the *expansion* of the instance $A \Leftrightarrow B$.

A fundamental property required of notational definitions is that they be formally nonessential [WR27,Qui53]. The use of notational definitions is justified only when all instances of defined notations can be eliminated. That is, all instances of defined notations occurring in any given expression can be expanded so that only primitive constructs in the language $\mathcal{L}$ remain. A definition instance will also be required to uniquely determine its expansion (modulo some notion of lexical equivalence) and that the result of expanding all definition instances in an expression be independent of the order in which these instances are expanded. Ideally, any mechanism implemented to support notational definition would allow users to maintain the illusion of extending the syntax of the object language while at the same time ensuring that instances of defined notation can always be eliminated.

Syntactic operators that bind variables complicate the formalization of notational definitions. Indeed, they are *the* complication. Such variable binding constructs appear in the languages used in all branches of mathematics and in the syntax of many programming languages. For example, in each of the familiar expressions

$$\textstyle\sum_{x=1}^{y} x^2, \quad \int_0^y e^x dx, \quad \forall x \in \mathcal{N}.x \leq y, \quad \{x \in \mathcal{N} | x \leq y\}$$

the variable x is *bound* while the variable y is *free*. The expression $x \leq y$ is said to be in the *scope* of the binding variable x in $\forall x \in \mathcal{N}.x \leq y$. To address complications involving binding variables, the operations of substitution, computation of free variables, and change of bound variables are identified as the *basic operations* on expressions.

Since defined notations can themselves contain binding variables, as in Equations 3.3—3.6, it should be possible to apply the basic operations directly to instances of defined notations without involving expansion. This is tantamount to requiring that defined notations have a discernible binding structure, and so implies that definitional equations must somehow specify the binding structure of defined notation. Figure 3.1 informally indicates the binding structure of the defined notations by annotating the schematic variables of the *lhs* with superscripts; the notation $v^{\vec{x}}$ is meant to indicate that v is in the scope of each variable in $\vec{x}$. In Equations 3.4, for example, the binding variables are represented by x and y; b is in the scope of x, while c is in the scope of both x and y. The conventional style of writing such definitions does not usually include this kind of annotation and leaves the binding structure implicit.

Binding variables also require that expansion of defined notation be quite different from simple macro expansion. To illustrate this, consider the following instance of the notation defined in Equation 3.2:

$$z \mid z$$

Surely this should not expand to

$$\exists z.z > 0 \land z \times z = z$$

but rather to

$$\exists w.w > 0 \land z \times w = z$$

for some new variable w. This problem, as well as many others, can be addressed by requiring that expansion commute with the basic operations on expressions. This is especially important with regard to the design of interfaces to interactive proof systems. In a "dialogue with a proof system," a user may enter an expression containing instances of defined notations that is subsequently involved in a sequence of operations. Requiring that expansion commute with the basic operations ensures that expanding any instances of defined notations in an expression before, during, or after a sequence of basic operations will result in an expression equivalent to the expression obtained from this sequence of operations without expansions.

Next, consider how expansion should handle the variable names used in an instance of defined notation. Suppose that a user of an interactive proof system enters (in some unspecified way) the following instance of the notation defined in Equation 3.4

$$\forall w \in A, u \in B.C$$

where w and u are variables and A, B, and C are expression. The user might be somewhat dismayed if expanding this definition resulted in

$$\forall z \in A.\forall v \in B[z/w].C[z, v/w, u]$$

where in the expansion, the binding variables w and u have been lost and are replaced by variables z and v. The user would most likely rather have the expansion result in

$$\forall w \in A.\forall u \in B.C$$

where the binding variables $\{w, u\}$ used in the expansion are those occurring in the definition instance. This is more important than might seem at first glance, especially in regard to interactive proof systems. Users often attach mnemonic significance to chosen variables names, which in turn helps in negotiating large and complicated proofs. Accommodating a user's expectation regarding the preservation of variable names would make interacting with such a system more natural. Although, as will be seen, to simultaneously accomplish this and the requirement that the basic operations be defined on instances of defined notations is no simple matter.

Finally, a good framework for notational definitions should lend itself to implementations that are not overly expensive computationally. The implementation of expressions, defined notations, and expansion procedures should be reasonably efficient.

In summary, the seven properties that an adequate framework for notational definitions should satisfy are enumerated here.

Property 1: Generality. We would like to proceed from a formal presentation of a language $\mathcal{L}$ to a formal account of notational definitions over $\mathcal{L}$.

Property 2: All instances of defined notations can be eliminated.

Property 3: The basic operations are defined on instances of defined notations.

Property 4: Expansion must commute with the basic operations.

Property 5: Supports an equational presentation of definitions.

Property 6: Variable names are preserved on expansion.

Property 7: Lends itself to an efficient implementation.

3.2 An outline of the solution

This section presents an informal account of the solution to the problem of modeling notational definitions. The next section provides the formal definitions.

A central thesis is that a definitional equation, such those presented in Figure 3.1, can be modeled by introducing a new syntactic constant into $\Lambda[\Sigma^{\mathcal{L}}, \mathcal{D}]$ following the same approach and conventions outlined in section 2.2 for defining the basic syntactic operators of $\mathcal{L}$. Furthermore, this new constant must be introduced with a δ-equation in order to allow for its elimination.

Let us first consider the simple case where the left-hand side of a defining equation does not contain any binding variables. For example, assuming a λ-representation of the first-order language presented in section 2.2, suppose that we want to represent the new notation introduced in the equation

$$a \Leftrightarrow b \stackrel{\text{def}}{=} (a \Rightarrow b) \wedge (b \Rightarrow a). \tag{3.7}$$

This could be accomplished with the introduction of a new constant $\Leftrightarrow$ of type

$$\tau = (\mathsf{o} \times \mathsf{o}) \to \mathsf{o}$$

that encodes instances of the notation as

$$\overline{A \Leftrightarrow B} \;\equiv\; \Leftrightarrow(\langle \overline{A}, \overline{B} \rangle) \tag{3.8}$$

with the δ-equation

$$\Leftrightarrow \;\stackrel{\delta}{=}\; \lambda z^{\mathsf{o}\times\mathsf{o}}.\wedge(\langle \Rightarrow(\langle \pi_1(z), \pi_2(z)\rangle), \Rightarrow(\langle \pi_2(z), \pi_1(z)\rangle)\rangle) \tag{3.9}$$

where the body of the λ-abstraction on the right of this equation is obtained from

$$\overline{(a \Rightarrow b) \wedge (b \Rightarrow a)} \quad = \quad \wedge(\langle \Rightarrow (\langle a, b \rangle), \Rightarrow (\langle b, a \rangle) \rangle), \tag{3.10}$$

by replacing a with $\pi_1(z)$ and b with$\pi_2(z)$. The instance in equation 3.8 can then be expanded with the reduction

$$\begin{aligned} \overline{A \Leftrightarrow B} \quad &\equiv \quad \Leftrightarrow (\langle \overline{A}, \overline{B} \rangle) \\ &\twoheadrightarrow \quad \wedge(\langle \Rightarrow (\langle \overline{A}, \overline{B} \rangle), \Rightarrow (\langle \overline{B}, \overline{A} \rangle) \rangle) \\ &\equiv \quad \overline{(A \Rightarrow B) \wedge (B \Rightarrow A)} \end{aligned} \quad .$$

Instead of introducing new notations with simple δ-equations, the approach presented below uses a new intermediate form called a Δ-equation for definitional equations. A Δ-equation is written

$$d(\sigma) \quad \stackrel{\Delta}{=} \quad e, \tag{3.11}$$

where d is a new constant and σ, called an s-term, is either a variable, a pair of s-terms, or a λ-abstraction whose body is an s-term. The s-term σ serves to specify the binding structure of the defined notation explicitly, as well as to name its components. Such a Δ-equation is translated into a δ-equation

$$d \quad \stackrel{\delta}{=} \quad \mathcal{F}_e^\sigma \tag{3.12}$$

using an expression-valued function $\mathcal{F}_e^\sigma$ defined below. A given expression will be called a σ-instance if it has the same type and "shape" as the s-term σ. An expression $d(I)$, where I is a σ-instance, will be called an instance of the Δ-equation given above — it represents an instance of the notation introduced by the Δ-equation. The expansion of a definition instance $d(I)$ is then effected simply by the reduction sequence (starting with a δ-reduction)

$$d(I) \quad \longrightarrow \quad \mathcal{F}_e^\sigma(I) \quad \twoheadrightarrow \quad e'.$$

Returning to the example from above

$$\Leftrightarrow (\langle a, b \rangle) \quad \stackrel{\Delta}{=} \quad \wedge(\langle \Rightarrow (\langle a, b \rangle), \Rightarrow (\langle b, a \rangle) \rangle) \tag{3.13}$$

the function $\mathcal{F}_e^\sigma$ will be defined so that equation 3.9 is obtained by this translation. More generally, for the case where σ does not contain any λ-abstractions, assume without loss of generality that

$$\sigma \quad \equiv \quad \langle v_1, \langle v_2, \langle \ldots, v_n \rangle \ldots \rangle.$$

The simple δ-equation can then be obtained by translation as

$$d \stackrel{\delta}{=} \lambda z.e\left\{ \frac{\pi_1(z)}{v_1} \quad \cdots \quad \frac{\pi_n(z)}{v_n} \right\} \tag{3.14}$$

so that

$$d(\langle e_1, \langle e_2, \langle \ldots, e_n \rangle \ldots \rangle) \twoheadrightarrow e' =_\alpha e\left[\frac{e_1}{v_1} \quad \cdots \quad \frac{e_n}{v_n} \right].$$

Note that this case corresponds to the AUTOMATH [vD80] δ-equation, which allows the defined constant to take parameters

$$d(v_1, \ldots, v_n) \stackrel{\delta}{=} e \tag{3.15}$$

where the v_i are pairwise distinct variables, and where the expansion is defined as

$$d(e_1, \ldots, e_n) \longrightarrow e\left[\frac{e_1}{v_1} \quad \cdots \quad \frac{e_n}{v_n} \right].$$

Now, turning to the more complicated case involving binding variables, consider the example

$$\forall x \in a, y \in b^x.c^{x,y} \stackrel{\text{def}}{=} \forall x \in a.\forall y \in b.c \tag{3.16}$$

Recall that this equation is meant to indicate the structure of the defined notation — that there are three subexpressions and two binding variables, indicated schematically by a, b, c, x, and y respectively, with a outside of the scope of x and y, b in the scope of x and outside the scope of y, and c in the scope of both x and y.

Before presenting a Δ-equation, a simple δ-equation for this definition is developed. This new syntactic construct can be encoded with the new constant $\forall_2$ of type $\tau = \mathbf{set} \times (\mathrm{i} \rightarrow \mathbf{set} \times (\mathrm{i} \rightarrow \mathrm{o}))$. In this way, the encoding map can be extended to account for instances of this defined notation with

$$\overline{\forall w \in A, u \in B.C} \equiv \forall_2(\langle \overline{A}, \lambda w.\langle \overline{B}, \lambda u.\overline{C} \rangle \rangle). \tag{3.17}$$

(Note again that the use of the pairing operator allows us to accomplish this representation without duplicating the variables w and u in the encoded instance.) This new constant can be introduced into $\mathcal{D}$ with a δ-equation

$$\forall_2 \stackrel{\delta}{=} \lambda z^\tau.\hat{e}$$

defined so that a reduction

$$\forall_2(\langle \overline{A}, \lambda w.\langle \overline{B}, \lambda u, \overline{C} \rangle \rangle) \longrightarrow (\lambda z^\tau.\hat{e})(\langle \overline{A}, \lambda w.\langle \overline{B}, \lambda u, \overline{C} \rangle \rangle) \twoheadrightarrow e'$$

$$\forall x \in a, y \in b^x.c^{x,y} \stackrel{\text{def}}{=} \forall x \in a.\forall y \in b.c \tag{3.22}$$

$$\forall_2(\langle a, \lambda x.\langle b, \lambda y.c\rangle\rangle) \stackrel{\Delta}{=} \forall_1(\langle a, \lambda x.\forall_1(\langle b, \lambda y.c\rangle)\rangle) \tag{3.23}$$

$$\forall_2 \stackrel{\delta}{=} \lambda z^{\tau}.\forall_1(\langle \Phi_a[z], \lambda x.\forall_1(\langle \Phi_b[z], \lambda y.\Phi_c[z]\rangle)\rangle) \tag{3.24}$$

Figure 3.2: A conventional equation with related Δ and δ equations.

results in an instance e' of the λ-representation of the right-hand side of the defining equation

$$\overline{\forall x \in a.\forall y \in b.c} \equiv \forall_1(\langle a, \lambda x.\forall_1(\langle b, \lambda y.c\rangle)\rangle). \tag{3.18}$$

Assume that a constant $\forall_1$ of type $(\text{set} \times (\text{i} \to \text{o})) \to \text{o}$ represents the new notation introduced in Equation 3.3

$$\forall x \in a.b^x \stackrel{\text{def}}{=} \forall x.(x \in a) \Rightarrow b$$

and that the encoding map is extended to account for this notation with

$$\overline{\forall w \in A.B} \equiv \forall_1(\langle \overline{A}, \lambda w.\overline{B}\rangle).$$

The expression $\hat{e}$ can be obtained by replacing each variable a, b, and c in expression 3.18 with expressions involving z that, loosely speaking, "take apart" the argument z on reduction while instantiating the results in the appropriate binding context. For example, replacing

$$a \text{ with } \Phi_a \equiv \pi_1(z), \tag{3.19}$$
$$b \text{ with } \Phi_b \equiv \pi_1(\pi_2(z)(x)), \tag{3.20}$$
$$c \text{ with } \Phi_c \equiv \pi_2(\pi_2(z)(x))(y) \tag{3.21}$$

in expression 3.18 results in δ-equation 3.24 of Figure 3.2. To check that this definition of $\forall_2$ makes sense, replace z in expressions 3.20–3.21 by

$$I \equiv \langle \overline{A}, \lambda w.\langle \overline{B}, \lambda u, \overline{C}\rangle\rangle$$

to obtain the reductions

$$\begin{array}{lll} \pi_1(I) & \twoheadrightarrow & \overline{A}, \\ \pi_1(\pi_2(I)(x)) & \twoheadrightarrow & \overline{B}[x/w], \\ \pi_2(\pi_2(I)(x))(y) & \twoheadrightarrow & \overline{C}[x/w][y/u], \end{array}$$

and then expand an instance of this notation by the reduction

$$\begin{aligned}
\overline{\forall w \in A, u \in B.C} &\equiv \forall_2(\langle \overline{A}, \lambda w.\langle \overline{B}, \lambda u.\overline{C}\rangle\rangle) \\
&\longrightarrow (\lambda z.\hat{e})(\langle \overline{A}, \lambda w.\langle \overline{B}, \lambda u.\overline{C}\rangle\rangle) \\
&\twoheadrightarrow \forall_1(\langle \overline{A}, \lambda x.\forall_1(\langle \overline{B}[x/w], \lambda y.\overline{C}[x/w][y/u]\rangle)\rangle) \\
&\equiv \overline{\forall x \in A.\forall y \in B[x/w].C[x/w][y/u]}.
\end{aligned}$$

It is assumed, for the sake of a clear presentation, that the variables $\{x, y\}$ are not α-converted in the process of reduction.

Instead of using a simple δ-equation to define this new notation, δ-equation 3.24 can be obtained by translation from Δ-equation 3.23. A Δ-equation is intended to be much closer to a conventional definitional equation than is a simple δ-equation. For example, given the encoding map discussed above, Δ-equation 3.23 can be viewed as

$$\overline{\forall x \in a, y \in b.c} \triangleq \overline{\forall x \in a.\forall y \in b.c}, \tag{3.25}$$

which is much closer to the defining equation 3.16 than is δ-equation 3.24. The relationship between conventional definitional equations and Δ-equations involves both the λ-representation of language constructs and a straightforward notational translation, and will be discussed along with other implementation concerns in Chapter 5.

It should be noted that the variables occurring in the left-hand side of a conventional defining equation, such as 3.16, are actually schematic *meta*-variables denoting locations for expression and variables, while those of the left hand side of the Δ-equation 3.25 are variables of Λ. It is hoped that this "pun" does not cause any confusion.

To provide for expansions that satisfy additional properties, define an *acceptable expansion* to be any expression valued function Exp such that $\mathrm{Exp}[d(I)]$ is definitionally equal to $d(I)$. One particular function Exp^{Δ}, is then defined, is shown to be an acceptable expansion, and is shown to satisfy Properties 6 and 7 by preserving variables on expansion in most cases, and by lending itself well to an efficient implementation.

This solution to the problem outlined at the outset of this chapter provides a clean separation of concerns by untangling Properties 1–7 in the following way. Property 1, generality, is addressed by the use of a Church-style encoding of a given object language $\mathcal{L}$. Properties 2–4, which require that defined notations be eliminable, that the basic operations be defined on instances of defined notations, and that expansion commute with the basic operations, are satisfied, respectively, by the representation of defining equations as δ-equations and of instances of defined notations as λ-expressions. The equational presentation of new notations, Property 5, is accomplished Δ-equations, which are translated into δ-equations. Additional properties, such as Properties 6 and 7, are addressed by defining acceptable expansions satisfying these properties.

3.3 Making it precise

First, define *schematic terms*, called s-terms for short, as follows:

Definition 7 (**s-terms**)

1. *Any variable* v^τ *is an s-term,*
2. *If* σ_1 *and* σ_2 *are s-terms and* $\mathrm{Vars}[\sigma_1] \cap \mathrm{Vars}[\sigma_2] = \phi$ *then* $\langle \sigma_1, \sigma_2 \rangle$ *is an s-term,*
3. *If* σ_1 *is an s-term and* $y \notin \mathrm{Vars}[\sigma_1]$ *then* $\lambda y.\sigma_1$ *is an s-term.*

The meta-variables $\sigma, \sigma_1, \sigma_2, \ldots$ will denote s-terms.

An expression having the "same shape" as an s-term σ is called a σ*-instance.*

Definition 8 (**σ-instances**) *Given an s-term* σ*, an expression* I *is called a* σ*-instance if one of the following holds:*

1. $\sigma \equiv v^\tau$ *and* $I \in \tau$*,*
2. $\sigma \equiv \langle \sigma_1, \sigma_2 \rangle$ *and* $I \equiv \langle I_1, I_2 \rangle$ *where* I_1 *is a* σ_1*-instance and* I_2 *is a* σ_2*-instance.*
3. $\sigma \equiv \lambda y^\tau.\sigma_1$ *and* $I \equiv \lambda x^\tau.I_1$ *where* I_1 *is a* σ_1*-instance.*

The meta-variables $I, I_1, I_2, \ldots$ will denote σ-instances.

For example,

$$\sigma \equiv \langle a^{\mathbf{set}}, \lambda x^{\mathbf{i}}.\langle b^{\mathbf{set}}, \lambda y^{\mathbf{i}}.c^{\mathbf{o}} \rangle \rangle \tag{3.26}$$

is an s-term and

$$I \equiv \langle \overline{A}, \lambda w^{\mathbf{i}}.\langle \overline{B}, \lambda u^{\mathbf{i}}.\overline{C} \rangle \rangle \tag{3.27}$$

is a σ instance where A and B are sets and C is a proposition in the first-order language.

Each s-term σ and each σ-instance I has a type that will be denoted as $\tau[\sigma]$.

Definition 9 (**$\tau[\sigma]$**) *Given an s-term* σ *and a variable* $v \in \mathrm{Evars}[\sigma]$*, define the type* $\tau[\sigma] \in \mathcal{T}$ *by induction on the structure of* σ*:*

$$\begin{aligned}
\tau[v^\tau] &\equiv \tau \\
\tau[\langle \sigma_1, \sigma_2 \rangle] &\equiv \tau[\sigma_1] \times \tau[\sigma_2] \\
\tau[\lambda y^{\tau'}.\sigma_1] &\equiv \tau' \rightarrow \tau[\sigma_1]
\end{aligned}$$

It is easy to show, for all s-terms σ and all σ-instances I, that $I \in \tau[\sigma]$. The s-term σ and the σ-instance I given in Equations 3.26 and 3.27 are both of type $\mathbf{set} \times (\mathbf{i} \rightarrow \mathbf{set} \times (\mathbf{i} \rightarrow \mathbf{o}))$.

Now define the expression $\Phi^\sigma_v[z]$ in such a way that, for any σ-instance I, the reduction of $\Phi^\sigma_v[z][I/z]$ takes apart I while instantiating λ-abstractions to the appropriate variables of σ.

Definition 10 (Φ_v^σ) *Given an s-term σ and a variable $v \in \mathrm{Evars}[\sigma]$ and any expression $e \in \tau[\sigma]$, define the expression $\Phi_v^\sigma[e]$ by induction on the structure of σ:*

$$\begin{array}{lll} \Phi_v^v[e] & \equiv e & \\ \Phi_v^{\langle\sigma_1,\sigma_2\rangle}[e] & \equiv \Phi_v^{\sigma_i}[\pi_i(e)] & (v \in \mathrm{Evars}[\sigma_i],\ i = 1 \text{ or } i = 2) \\ \Phi_v^{\lambda y.\sigma_1}[e] & \equiv \Phi_v^{\sigma_1}[e(y)] & \end{array}$$

From the s-term σ defined in Equation 3.26, the following expressions are obtained

$$\begin{aligned} \Phi_a^\sigma &\equiv \pi_1(z), \\ \Phi_b^\sigma &\equiv \pi_1(\pi_2(z)(x)), \\ \Phi_c^\sigma &\equiv \pi_2(\pi_2(z)(x))(y). \end{aligned}$$

Note that these are exactly expressions 3.20–3.21 from above.

Definition 11 ($\mathcal{F}_e^\sigma$) *For any s-term σ with $\mathrm{Evars}[\sigma] = \{v_1 \ldots v_n\}$ and any expression e, define*

$$\mathcal{F}_e^\sigma \equiv \lambda z^{\tau[\sigma]}.e\left\{ \frac{\Phi_{v_1}^\sigma[z]}{v_1} \quad \cdots \quad \frac{\Phi_{v_n}^\sigma[z]}{v_n} \right\},$$

for $z = \mathrm{NewVar}[\mathrm{Vars}[\sigma] \cup \mathrm{Vars}[e]]$.

To ensure that $\mathcal{F}_e^\sigma$ is well-formed, require that e be *compatible with σ*.

Definition 12 (e **is compatible with** σ) *An expression e is said to be* compatible *with an s-term σ if*

1. $\mathrm{Fvars}[\sigma]$ $(= \mathrm{Evars}[\sigma])$ $= \mathrm{Fvars}[e]$,
2. $\mathrm{Bvars}[\sigma] \subseteq \mathrm{Bvars}[e]$,
3. $\mathrm{Evars}[\sigma] \cap \mathrm{Bvars}[e] = \phi$,
4. *if $v \in \mathrm{Evars}[\sigma]$ is in the scope of $y \in \mathrm{Bvars}[\sigma]$ in σ then every occurrence of v in e is in the scope of some occurrence of y in e,*
5. *the type associated with each variable in $\mathrm{Vars}[\sigma]$ matches the type associated with the corresponding variable in e.*

Lemma 1 *If e is compatible with σ then*

1. $\mathrm{Fvars}[\mathcal{F}_e^\sigma] = \phi$.
2. *$\mathcal{F}_e^\sigma$ is well formed.*

It is now possible to define Δ-equations:

Definition 13 (Δ-equations) *Given an s-term σ and an expression e compatible with σ and a new constant d*

1. *The equation*
$$d(\sigma) \ \stackrel{\Delta}{\equiv}\ e$$
is called a Δ-equation, or the defining Δ-equation for d.

2. *The meaning of such a Δ-equation is given by the δ-equation*
$$d \ \stackrel{\delta}{=}\ \mathcal{F}_e^\sigma$$
which is a well formed δ-equation by Lemma 1.

3. *If I is a σ-instance the expression $d(I)$ is called an instance of the notation introduced by the defining Δ-equation for d.*

As a trivial example, it is easy to check that for any s-term σ

$$d(\sigma) \ \stackrel{\Delta}{\equiv}\ \sigma \tag{3.28}$$

is a well-formed Δ-equation. In addition,

Fact 1 *If σ is an s-term and I is an σ-instance then*

$$\mathcal{F}_\sigma^\sigma(I) \ \twoheadrightarrow\ I.$$

This is exactly what should be expected from Equation 3.28.

Returning to the example given above, from Δ-equation 3.23 of Figure 3.2, the expression $\mathcal{F}_e^\sigma$ yields δ-equation 3.24 so that

$$\overline{\forall w \in A, u \in B.C} \ \twoheadrightarrow\ \overline{\forall x \in A.\forall y \in B[x/w].C[x/w][y/u]}.$$

Figure 3.3 presents Equations 3.1-3.5 of Figure 3.1 together with their corresponding Δ-equations.

Extend the notation introduced in Chapter 2 for lists of δ-equations $\mathcal{D}$ to contain not only simple δ-equations, but also Δ-equations $d(\sigma) \stackrel{\Delta}{\equiv} e$ representing the δ-equation $d \stackrel{\delta}{=} \mathcal{F}_e^\sigma$.

3.4 Acceptable expansions.

It may be desirable to expand instances $d(I)$ of defined notations in some way other than the reduction of $\mathcal{F}_e^\sigma(I)$, and so the notion of a function Exp being an *acceptable expansion* is defined.

$a \Leftrightarrow b$	$\stackrel{def}{=}$	$(a \Rightarrow b) \wedge (b \Rightarrow a)$
$\Leftrightarrow (\langle a, b \rangle)$	$\triangleq$	$\wedge(\langle \Rightarrow (\langle a, b \rangle), \Rightarrow (\langle b, a \rangle), \rangle)$
$a \mid b$	$\stackrel{def}{=}$	$\exists z.z > 0 \wedge a \times z = b$
$\mid (\langle a, b \rangle)$	$\triangleq$	$\exists(\lambda z. \wedge (\langle > (\langle z, 0 \rangle), = (\langle \times(\langle a, z \rangle), b \rangle)\rangle))$
$\forall x \in a.b^x$	$\stackrel{def}{=}$	$\forall x.(x \in a) \Rightarrow b$
$\forall_1(\langle a, \lambda x.b)$	$\triangleq$	$\forall(\lambda x. \Rightarrow (\langle \in (\langle x, a \rangle), b \rangle)$
$\forall x \in a, y \in b^x.c^{x,y}$	$\stackrel{def}{=}$	$\forall x \in a.\forall y \in b.c$
$\forall_2(\langle a, \lambda x.\langle b, \lambda y.c \rangle \rangle)$	$\triangleq$	$\forall_1(\langle a, \lambda x.\forall_1(\langle b, \lambda y.c \rangle)\rangle)$
$\forall x, y \in a.b^{x,y}$	$\stackrel{def}{=}$	$\forall x \in a.\forall y \in a.b$
$\forall_2'(\langle a, \lambda x.\lambda y.b \rangle)$	$\triangleq$	$\forall_1(\langle a, \lambda x.\forall_1(\langle a, \lambda y.b \rangle)\rangle)$
$\exists! x.a^x$	$\stackrel{def}{=}$	$\exists x.a \wedge (\forall y.a[y/x] \Rightarrow x = y)$
$\exists!(\lambda x.a)$	$\triangleq$	$\exists(\lambda x. \wedge (\langle a, \forall(\lambda y. \Rightarrow (\langle(\lambda x.a)(y), = (\langle x, y \rangle)\rangle))\rangle))$

Figure 3.3: Defining equations with corresponding Δ-equations.

Definition 14 (Acceptable Expansion)
An expression valued function $\mathrm{Exp}_{\mathcal{D}}[d(I)]$ *is called an acceptable expansion if, for every* Δ*-equation*

$$d(\sigma) \;\stackrel{\Delta}{\equiv}\; e$$

and every σ*-instance* I,

$$d(I) \;=_{\Lambda}\; \mathrm{Exp}_{\mathcal{D}}[d(I)].$$

We will usually write Often the expression $\mathrm{Exp}_{\mathcal{D}}[d(I)]$ is written as $\mathrm{Exp}[d(I)]$ when the set $\mathcal{D}$ can be inferred from context.

The definition of $\mathcal{F}^{\sigma}_{e}$ can now be viewed as providing a correctness criterion for acceptable expansions. Also, it is easy to check that any acceptable expansion commutes with the basic operations on terms, and so satisfies Property 4.

Lemma 2 *If* Exp *is an acceptable expansion, then for every* Δ*-equation*

$$d(\sigma) \;\stackrel{\Delta}{\equiv}\; e$$

and every σ*-instance* I,

1. $\mathrm{Fvars}[d(I)] = \mathrm{Fvars}[\mathrm{Exp}[d(I)]]$,
2. $\mathrm{Exp}[d(I)[e_1/v_1,\ldots,e_n/v_n]] =_{\Lambda} \mathrm{Exp}[d(I)][e_1/v_1,\ldots,e_n/v_n]$, *and*
3. *if* $I =_{\Lambda} I'$ *then* $\mathrm{Exp}[d(I)] =_{\Lambda} \mathrm{Exp}[d(I')]$.

For example, it is easy to see that the functions

$$\begin{aligned}
\mathrm{Exp}^{id}[d(I)] &\equiv d(I)\\
\mathrm{Exp}^{1}[d(I)] &\equiv \mathcal{F}^{\sigma}_{e}(I)\\
\mathrm{Exp}^{\mathrm{nf}}[d(I)] &\equiv \mathrm{nf}^{\beta\eta\pi\delta}[d(I)]
\end{aligned}$$

are acceptable expansions, although Exp^{id} does not really perform any expansion. It is expected that the application of most interesting expansion functions will result in an expression α-equal to some expression obtained by reducing $d(I)$:

$$d(I) \;\longrightarrow\; \mathcal{F}^{\sigma}_{e}(I) \;\twoheadrightarrow\; e' \;=_{\alpha}\; \mathrm{Exp}_{\mathcal{D}}[d(I)]$$

Indeed, this could have taken as defining an acceptable expansion, but it turns out that this makes it more difficult to prove that certain expansion functions are acceptable. Such an acceptable expansion, Exp^{Δ}, will be defined in Chapter 4. This expansion preserves variable names in most instances.

3.5 Notational definition over a language $\mathcal{L}$

Of course, not all Δ-equations will correspond to definitions over an encoded language $\mathcal{L}$. For example, returning to the encoding of a first-order language Σ^{fol}, the Δ-equation

$$\text{nope}(\lambda y^{\mathbf{i}\to\mathbf{i}}.v^{\mathbf{o}}) \;\stackrel{\Delta}{=}\; \lambda y^{\mathbf{i}\to\mathbf{i}}\langle v^{\mathbf{o}}, v^{\mathbf{o}}\rangle$$

probably would not be considered as an equation representing the definition of new notation over (the encoding of) the first-order language. This is not simply because the right-hand side of this equation does not "look like" a first-order expression because the notation

$$p \text{ where } x \text{ is } a \;\stackrel{\text{def}}{=}\; p[a/x]$$

can be represented with the Δ-equation

$$\text{where}(\langle a^{\mathbf{i}}, \lambda x^{\mathbf{i}}.p^{\mathbf{o}}\rangle) \;\stackrel{\Delta}{=}\; (\lambda x.p)(a)$$

and this seems like a perfectly reasonable notational extension of the first-order language, as long as expanding an instance of this notation involves reducing the β-redex of the right-hand side.

Which Δ-equations, then, correspond to notational extensions of $\mathcal{L}$? Given an encoding of $\mathcal{L}$, define notational extensions to $\overline{\mathcal{L}}$ by restricting the possible *types* of a Δ-equation's right-hand side. Also, define an acceptable expansion to be acceptable for a notational extensions of $\overline{\mathcal{L}}$ if the result of expanding any instance of defined notation is in $\beta\eta\pi$-normal form.

Suppose that $\mathcal{L}$ is a language encoded in the manner outlined in Section 2.2, with the signature $\Sigma^{\mathcal{L}} = (T_0^{\mathcal{L}}, S^{\mathcal{L}})$, so that

$$\overline{\mathcal{L}} \;=\; \bigcup_{t\in T_0^{\mathcal{L}}} \mathcal{NF}^{\beta\eta\pi}[\Sigma^{\mathcal{L}}, [], \{t\}, T_t] \qquad (3.29)$$

where each T_t is a subset of $T_0^{\mathcal{L}}$. Then, given a collection of δ- and Δ-equations $\mathcal{D}$, the set

$$\overline{\mathcal{L}}^{\mathcal{D}} \;=\; \bigcup_{t\in T_0^{\mathcal{L}}} \mathcal{NF}^{\beta\eta\pi}[\Sigma^{\mathcal{L}}, \mathcal{D}, \{t\}, T_t].$$

corresponds to a notational extension of $\overline{\mathcal{L}}$, since

Fact 2 *If* $e \in \overline{\mathcal{L}}^{\mathcal{D}}$, *then* $\text{nf}^{\beta\eta\pi\delta}[e] \in \overline{\mathcal{L}}$.

However, such a $\mathcal{D}$ may still contain defined notations that do not correspond to notational extensions of $\mathcal{L}$. To address this, define only certain $\mathcal{D}$ to be *notational extensions of* $\overline{\mathcal{L}}$.

Definition 15 ($\mathcal{D}$ is a notational extension of $\overline{\mathcal{L}}$) *First, given the description of $\overline{\mathcal{L}}$ in equation 3.29, define for any $\mathcal{D}$ the set $\overline{\mathcal{L}}^{\mathcal{D}+}$ as*

$$\overline{\mathcal{L}}^{\mathcal{D}+} = \bigcup_{t \in T_0^{\mathcal{L}}} \Lambda[\Sigma^{\mathcal{L}}, \mathcal{D}, \{t\}, T_t].$$

Then define the set of notational extensions *of $\overline{\mathcal{L}}^{\mathcal{D}+}$ by induction as follows.*

Then define to be a notational extension of , by induction on the length of $\mathcal{D}$.

1. *[] is a notational extension of $\overline{\mathcal{L}}$,*
2. *if $\mathcal{D}$ is a notational extension of $\overline{\mathcal{L}}$ and $\mathcal{D}[d \overset{\delta}{=} e]$ is a well-formed list of δ- and Δ-equations with $e \in \overline{\mathcal{L}}^{\mathcal{D}}$ then $\mathcal{D}[d \overset{\delta}{=} e]$ is a notational extension of $\overline{\mathcal{L}}$,*
3. *if $\mathcal{D}$ is a notational extension of $\overline{\mathcal{L}}$ and $\mathcal{D}[d(\sigma) \overset{\Delta}{=} e]$ is a well-formed list of δ- and Δ-equations with $e \in \overline{\mathcal{L}}^{\mathcal{D}+}$ then $\mathcal{D}[d(\sigma) \overset{\Delta}{=} e]$ is a notational extension of $\overline{\mathcal{L}}$.*

This is meant to capture the intuitive notion that each equation in $\mathcal{D}$ models a notational definition over $\mathcal{L}$.

Definition 16 (an acceptable expansion for $\overline{\mathcal{L}}^{\mathcal{D}}$)
Given a notational extension $\mathcal{D}$ of $\overline{\mathcal{L}}$, define an acceptable expansion Exp *to be an* acceptable expansion for $\overline{\mathcal{L}}^{\mathcal{D}}$ *if, for every instance of Δ-defined notation $d(I) \in \overline{\mathcal{L}}^{\mathcal{D}}$,* $\mathrm{Exp}[d(I)] \in \overline{\mathcal{L}}^{\mathcal{D}}$.

Any acceptable expansion naturally gives rise to an acceptable expansion for $\overline{\mathcal{L}}^{\mathcal{D}}$:

Lemma 3 *If* $\mathrm{Exp}[d(I)]$ *is any acceptable expansion, then*

$$\mathrm{Exp}'[d(I)] \equiv \mathrm{nf}^{\beta\eta\pi}[\mathrm{Exp}[d(I)]]$$

is an acceptable expansion for $\overline{\mathcal{L}}^{\mathcal{D}}$.

3.6 Iterative notations

Although this framework models a large class of notational definitions, it does not account for many conventional abbreviations that have a natural *iterative* description. For example, the definitions given in Equations 3.3 and 3.4 suggests a family of notations for n-ary universal quantifiers:

$$\begin{aligned}
\forall x_1 \in a_1.b &\stackrel{\text{def}}{=} \forall x_1.(x_1 \in a_1) \Rightarrow b \\
\forall x_1 \in a_1, x_2 \in a_2.b &\stackrel{\text{def}}{=} \forall x_1 \in a_1.\forall x_2 \in a_2.b \\
\forall x_1 \in a_1, x_2 \in a_2, x_3 \in a_3.b &\stackrel{\text{def}}{=} \forall x_1 \in a_1.\forall x_2 \in a_2, x_3 \in a_3.b \\
&\vdots
\end{aligned}$$

where equation $i+1$, $i \geq 1$, uses the notation defined in equation i in the expression on its right-hand side.

As it stands, the framework can model these definitions by introducing one Δ-equation for each of the above equations:

$$\begin{aligned}
\forall_1(\langle a_1, \lambda x_1.b\rangle) &\triangleq \overline{\forall x_1.(x_1 \in a_1) \Rightarrow b} \\
\forall_2(\langle a_1, \lambda x_1.\langle a_2, \lambda x_2.b\rangle\rangle) &\triangleq \overline{\forall x_1 \in a_1.\forall x_2 \in a_2.b} \\
\forall_3(\langle a_1, \lambda x_1.\langle a_2, \lambda x_2.\langle a_3, \lambda x_3.b\rangle\rangle\rangle) &\triangleq \overline{\forall x_1 \in a_1.\forall x_2 \in a_2, x_3 \in a_3.b} \\
&\vdots
\end{aligned}$$

Clearly, this is not satisfactory. Alternatively, we would like to define these new notations using *only two* equations: a base case and an inductive case:

$$\begin{aligned}
\forall_1(\langle a_1, \lambda x_1.b\rangle) &\triangleq \overline{\forall x_1.(x_1 \in a_1) \Rightarrow b} \\
\forall_{n+1}(\langle a_1, \lambda x_1.s\rangle) &\triangleq \forall_1(\langle a_1, \lambda x_1.\forall_n(s)\rangle)
\end{aligned}$$

where s is a new kind of meta-variable ranging over s-terms.

This is exactly the solution proposed here. First, extend the set of constants $\mathcal{C}$ to include constants $d_{(m)}$ having integer subscripts. Define a new class of variables $\mathcal{VS}$ ranging over s-terms. Then define an *extended s-term* as follows:

1. any variable v^τ is an extended s-term,

2. any $s \in \mathcal{VS}$ is an extended s-term,

3. If σ_1 and σ_2 are extended s-terms, $\text{Vars}[\sigma_1] \cap \text{Vars}[\sigma_2] = \phi$, and any $\mathcal{VS}$ variable occurring in either s-term is the same variable, then $\langle \sigma_1, \sigma_2\rangle$ is an extended s-term,

4. If σ_1 is an extended s-term and $y \notin \text{Vars}[\sigma_1]$ then $\lambda y.\sigma_1$ is an s-term.

An extended s-term that contains instances of a $\mathcal{VS}$ variable s will be written as $\sigma[s]$ (by definition, s is the unique $\mathcal{VS}$ variable in $\sigma[s]$). If σ_1 is an s-term and $\sigma[s]$ is an extended s-term, then $\sigma[\sigma_1']$ denotes some s-term that is obtained by first changing any variables in σ_1 to obtain σ_1' in such a way as to avoid any repetition of variable names from $\sigma[s]$, and then replacing all occurrences of s in $\sigma[s]$ by σ_1'.

Define the declaration an *iterative notation* to be given by a pair of Δ-equations of the form

$$d_{(1)}(\sigma_1) \ \stackrel{\Delta}{=}\ e_1 \tag{3.30}$$

$$d_{(n+1)}(\sigma[s]) \ \stackrel{\Delta}{=}\ e[d_n(s)] \tag{3.31}$$

where $e[d_n(s)]$ indicates that there is at least one, and possibly several, occurrences of $d_n(s)$ in e, and the meta-variable n is ranging over integers. Such a pair is *well-formed* if the first Δ-equation is well-formed and if the equation

$$d_{(n+1)}(\sigma[v^\tau]) \ \stackrel{\Delta}{=}\ e[v^\tau]$$

is well-formed, where v is a new variable and τ is the type of e_1.

Together, Equations 3.30 and 3.31 can be taken to represent the family of Δ-equations

$$\begin{array}{lll} d_{(1)}(\sigma_1) & \stackrel{\Delta}{=} & e_1 \\ d_{(2)}(\sigma[\sigma_1']) & \stackrel{\Delta}{=} & e[d_1(\sigma_1')] \\ d_{(3)}(\sigma[\sigma'[\sigma_1']]) & \stackrel{\Delta}{=} & e[d_2(\sigma'[\sigma_1'])] \\ & \vdots & \end{array}$$

where σ_1', $\sigma'[\sigma_1']$, etc., indicate that the variables of these s-terms have been changed to avoid any repetition of variable names. It is easy to check that if the original pair is well-formed in the sense defined above, then each Δ-equation in this family is well-formed.

Extend the notation for lists of δ-equations $\mathcal{D}$ to contain not only δ-equations $d \stackrel{\delta}{=} e$ and Δ-equations $d(\sigma) \stackrel{\Delta}{=} e$, but also pairs of the form

$$(d_{(1)}(\sigma_1) \stackrel{\Delta}{=} e_1,\ d_{(n+1)}(\sigma[s]) \stackrel{\Delta}{=} e[d_n(s)]).$$

For the purposes of type checking and reduction, such pairs need not be considered as representing an infinite list of Δ-equations, but as representing only a finite initial segment of such a list that is long enough to account for any instances $d_{(j)}(I)$ under consideration.

Chapter 4

The expansion Exp^{Δ}

Expanding $\overline{\forall w \in A, u \in B.C}$ in the example given in Section 3.3 resulted in the loss of the binding variables $\{w, u\}$. In this chapter one particular acceptable expansion, Exp^{Δ}, is defined to provide for the preservation of variable names (Property 6) in most cases. This expansion also lends itself to a relatively efficient implementation (Property 7). Achieving a "macro-like" expansion that is also acceptable is not easy. First, two new operations are defined, *generalized substitution* and *simultaneous α-conversion* in Sections 4.1 and 4.2, which work together to accomplish this. A proof that Exp^{Δ} is indeed an acceptable expansion is presented in Section 4.4.

4.1 Generalized substitution

Given an expression e, expressions $e_1, \ldots, e_n$, sets of variables $S_1, \ldots, S_n$ with $S_i \subseteq \mathrm{Fvars}[e_i]$, and pairwise distinct variables $v_1, \ldots, v_n$, the notation

$$e\left[\begin{array}{ccc} \frac{(e_1, S_1)}{v_1} & \cdots & \frac{(e_n, S_n)}{v_n} \end{array}\right]$$

will denote the simultaneous substitution of e_i for all free occurrences of v_i in e, allowing the possibility of capture in e_i of variables in S_i. The notation

$$e\left[\begin{array}{ccc} \frac{(e_1, S_1)}{v_1} & \cdots & \frac{(e_n, S_n)}{v_n} \end{array}\right] / \{v_j\}$$

denotes substitution with the j^{th} component of the domain deleted. Also, assume a function $\mathrm{NewVar}[S]$ that takes a finite set of variables S and returns a new variable $z \notin S$.

$$
\begin{array}{ll}
c^* & \equiv c \\
v^* & \equiv v \quad (\text{if } v \notin \{v_1 \ldots v_n\}) \\
 & \equiv e_i \quad (\text{if } v \equiv v_i) \\
\langle e', e'' \rangle^* & \equiv \langle e'^*, e''^* \rangle \\
(\pi_i(e'))^* & \equiv \pi_i(e'^*) \quad (i \in \{1,2\}) \\
(e'(e''))^* & \equiv e'^*(e''^*) \\
(\lambda x.e')^* & \\
 & \text{Case 1: } n = 1 \text{ and } x \equiv v_1 \\
 & \equiv \lambda x.e' \\
 & \text{Case 2: } n > 1 \text{ and } x \equiv v_j \\
 & \equiv (\lambda x.e') \left[\dfrac{(e_1, S_1)}{v_1} \; \ldots \; \dfrac{(e_n, S_n)}{v_n} \right] / \{v_j\} \\
 & \text{Case 3: } x \notin \vec{v} \text{ and } x \notin \bigcup_{1 \leq i \leq n}(\text{Fvars}[e_i] - S_i) \\
 & \equiv \lambda x.e'^* \\
 & \text{Case 4: } x \notin \vec{v} \text{ and } x \in \bigcup_{1 \leq i \leq n}(\text{Fvars}[e_i] - S_i). \\
 & \equiv \lambda z.(e' \left[\dfrac{z}{x} \right])^* \\
 & \text{where } z = \text{NewVar}[\text{Vars}[e'] \cup \{\vec{v}\} \cup (\bigcup_{1 \leq i \leq n}(\text{Fvars}[e_i] - S_i))]
\end{array}
$$

Figure 4.1: Simultaneous capture-incurring substitution.

Definition 17 (simultaneous capture-incurring substitution)
Let $v_1, \ldots v_n$ be n ($n > 0$) be pairwise distinct variables, $e_1, \ldots e_n$ be expressions, and $S_1, \ldots S_n$ be sets of variables such that $S_i \subseteq \text{Fvars}[e_i]$. Let

$$e^* \;=\; e\left[\;\frac{(e_1, S_1)}{v_1}\;\cdots\;\frac{(e_n, S_n)}{v_n}\;\right]$$

be an abbreviation for the simultaneous capture-incurring substitution of e_i for v_1 in e that is defined by induction on the structure of e and the integer n in Figure 4.1.

This is called *generalized substitution* since the standard notion of simultaneous replacement of free occurrences of v_i by e_i in e.

$$e\left\{\;\frac{e_1}{v_1}\;\cdots\;\frac{e_n}{v_n}\;\right\}.$$

can be viewed as representing the special case where $S_i = \text{Fvars}[e_i]$, while the standard notion of simultaneous capture-avoiding substitution

$$e\left[\;\frac{e_1}{v_1}\;\cdots\;\frac{e_n}{v_n}\;\right].$$

can be viewed as representing the special case where $S_i = \phi$ ($i = 1, \ldots, n$).

The following facts concerning simultaneous capture-avoiding substitution will be useful in proofs presented below.

Fact 3
For any expression e and any $n \geq 1$ and expressions $e_1, \ldots, e_n$ and $e'_1, \ldots, e'_n$, if $e_i \twoheadrightarrow e_i$ for $i \in \{1, \ldots, n\}$, then

$$e\left[\;\frac{e_1}{v_1}\;\cdots\;\frac{e_n}{v_n}\;\right] \;\twoheadrightarrow\; e\left[\;\frac{e'_1}{v_1}\;\cdots\;\frac{e'_n}{v_n}\;\right]$$

Fact 4 *For any expression e and e' with $e =_\alpha e'$ and expressions $e_1, \ldots, e_n$ ($n \geq 1$),*

$$e\left[\;\frac{e_1}{v_1}\;\cdots\;\frac{e_n}{v_n}\;\right] \;=_\alpha\; e'\left[\;\frac{e_1}{v_1}\;\cdots\;\frac{e_n}{v_n}\;\right]$$

Fact 5 *Let $\vec{y} = [y_1, \ldots, y_n]$ be pairwise distinct variables and e be any expression. If $y_i \notin \text{Fvars}[e]$ then $e[\vec{e}/\vec{y}] =_\alpha e[\vec{e}/\vec{y}]/\{y_i\}$.*

Fact 6 *Let $\vec{y} = [y_1, \ldots, y_n]$ and $\vec{v} = [v_1, \ldots, v_n]$ be pairwise distinct variables, and let $\vec{e_1} = [e_{1,1}, \ldots, e_{1,n}]$ and $\vec{e_2} = [e_{2,1}, \ldots, e_{2,n}]$ be expressions. Then*

$$e[\vec{e_1}/\vec{y}][\vec{e_2}/\vec{v}] =_\alpha e[\vec{e_3}/\vec{w}]$$

where

$$\begin{aligned} \vec{e_3} &= [e_{1,1}[\vec{e_2}/\vec{v}], \ldots, e_{1,n}[\vec{e_2}/\vec{v}], e_{2,1}, \ldots, e_{2,n}], \\ \vec{w} &= [y_1, \ldots, y_n, v_1, \ldots, v_n]. \end{aligned}$$

$$
\begin{array}{ll}
c[\vec{w}/\vec{v}]^\alpha & \equiv c \\
v[\vec{w}/\vec{v}]^\alpha & \equiv v \\
\langle e_1, e_2\rangle[\vec{w}/\vec{v}]^\alpha & \equiv \langle e_1[\vec{w}/\vec{v}]^\alpha, e_2[\vec{w}/\vec{v}]^\alpha\rangle \\
(\pi_i(e_1))[\vec{w}/\vec{v}]^\alpha & \equiv \pi_i(e_1[\vec{w}/\vec{v}]^\alpha) \quad (i \in \{1,2\}) \\
(e_1(e_2))[\vec{w}/\vec{v}]^\alpha & \equiv e_1[\vec{w}/\vec{v}]^\alpha(e_2[\vec{w}/\vec{v}]^\alpha) \\
(\lambda x.e_1)[\vec{w}/\vec{v}]^\alpha & \\
& \text{Case 1: } x \notin \{\vec{v}\} \cup \{\vec{w}\} \\
& \equiv \lambda x.e_1[\vec{w}/\vec{v}]^\alpha \\
& \text{Case 2: } x \equiv v_j \\
& \equiv (\lambda w_j.e_1[\vec{w}/\vec{v}]^\alpha[w_j/v_j]) \\
& \text{Case 3: } x \equiv w_j \\
& \equiv (\lambda z.e_1[z/x][\vec{w}/\vec{v}]^\alpha) \\
& \quad \text{where } z = \text{NewVar}[\text{Vars}[e_1] \cup \{\vec{v}\} \cup \{\vec{w}\}]
\end{array}
$$

Figure 4.2: Simultaneous α-conversion.

4.2 Simultaneous α-conversion

To address the preservation of variable names it must be possible to insert, in a secure manner, a collection of binding variables into designated target locations in a given expression. To this end, simultaneous α-conversion is defined.

Definition 18 (simultaneous α-conversion)
Let e be any expression. Given pairwise distinct variables

$$\vec{v} \equiv [v_1, v_2, \ldots v_n]$$

and variables (not necessarily distinct)

$$\vec{w} \equiv [w_1, w_2, \ldots w_n]$$

with $\{\vec{v}\} \cap \{\vec{w}\} = \phi$ *and* $\text{Fvars}[e] \cap \{\vec{w}\} = \phi$, *define the simultaneous alpha-replacement of* v_i *by* w_i *in e, written*

$$e[\vec{w}/\vec{v}]^\alpha,$$

by induction on the structure of e as in Figure 4.2.

Notice that, in the second case of the last clause, the condition $\text{Fvars}[e_1] \cap \{\vec{w}\} = \phi$ holds since, by assumption, $\text{Fvars}[\lambda x.e_1] \cap \{\vec{w}\} = \phi$, so $\text{Fvars}[e_1] \subseteq (\text{Fvars}[e_1] \cup \{x\})$, but in this case $x \notin \{\vec{w}\}$.

The following lemma shows that simultaneous α-conversion is well defined.

Lemma 4 *Let e be an expression. Given any $n \geq 1$ and variables*

$$\begin{array}{rcl} \vec{v} & \equiv & [v_1, v_2, \dots v_n] \\ \vec{w} & \equiv & [w_1, w_2, \dots w_n] \end{array}$$

if $\{\vec{v}\} \cap \{\vec{w}\} = \phi$ and $\text{Fvars}[e] \cap \{\vec{w}\} = \phi$ then

$$e[\vec{w}/\vec{v}]^\alpha \ =_\alpha \ e.$$

Proof. By induction on the structure of e. All cases are trivial, except for $e \equiv \lambda x.e_1$, which involves looking at three subcases:
Case 1: $x \notin \{\vec{v}\} \cup \{\vec{w}\}$. Since, by assumption, $\text{Fvars}[\lambda x.e_1] \cap \{\vec{w}\} = \phi$ and $x \notin \{\vec{w}\}$ it follows that $\text{Fvars}[e_1] \cap \{\vec{w}\} = \phi$, which allows the use of the induction hypothesis on the expression $e_1[\vec{w}/\vec{v}]^\alpha$ in

$$\begin{array}{rcl} (\lambda x.e_1)[\vec{w}/\vec{v}]^\alpha & \equiv & \lambda x.e_1[\vec{w}/\vec{v}]^\alpha \\ & =_\alpha & \lambda x.e_1 \quad (e_1[\vec{w}/\vec{v}]^\alpha =_\alpha e_1 \text{ by induction}) \\ & \equiv & e. \end{array}$$

Case 2: $x \notin \{\vec{w}\}$ and $x = v_j$ for some $j \in \{1, \dots, n\}$. Again, the induction hypothesis can be used on the expression $e_1[\vec{w}/\vec{v}]^\alpha$ (for the same reasons as cited in Case 1) in

$$\begin{array}{rcl} (\lambda x.e_1)[\vec{w}/\vec{v}]^\alpha & \equiv & (\lambda w_j.e_1[\vec{w}/\vec{v}]^\alpha[w_j/x]) \\ & =_\alpha & \lambda w_j.e_1[w_j/x] \quad (e_1[\vec{w}/\vec{v}]^\alpha =_\alpha e_1 \text{ by induction}) \\ & =_\alpha & \lambda x.e_1 \\ & \equiv & e. \end{array}$$

Case 3: $x \notin \{\vec{v}\}$ and $x = w_j$ for some $j \in \{1, \dots, n\}$. By assumption, $\text{Fvars}[\lambda x.e_1] \cap \{\vec{w}\} = \phi$, so if z is a new variable with $z \notin \text{Fvars}[e_1] \cup \{\vec{w}\}$, so $\text{Fvars}[e_1[z/x]] \cap \{\vec{w}\} = \phi$. The induction hypothesis can now be used on the expression $e_1[z/x]$ in

$$\begin{array}{rcl} (\lambda x.e_1)[\vec{w}/\vec{v}]^\alpha & \equiv & (\lambda z.e_1[z/x][\vec{w}/\vec{v}]^\alpha) \\ & =_\alpha & \lambda z.e_1[z/x] \\ & & (e_1[z/x][\vec{w}/\vec{v}]^\alpha =_\alpha e_1[z/x] \text{ by induction}) \\ & =_\alpha & \lambda x.e_1 \\ & \equiv & e. \end{array}$$

□

So for example, if $\vec{v} \equiv [x, y]$ and $\vec{w} \equiv [u, t]$ then

$$(\lambda u.\lambda x.\lambda y.c(u)(x)(y))[\vec{w}/\vec{v}]^{\alpha} \equiv \lambda z.\lambda u.\lambda t.c(z)(u)(t)$$

where z is a new variable.

4.3 Definition of Exp^{Δ}

Next, assume that the set of variables $\mathcal{V}$ is partitioned into two sets,

$$\mathcal{V} = \mathcal{V}_s \cup \mathcal{V}_r$$

with

$$\mathcal{V}_s \cap \mathcal{V}_r = \phi.$$

The set $\mathcal{V}_s$ is called the set of *schematic-variables*, while the set $\mathcal{V}_r$ is called the set of *regular-variables.* Also assume, for every s-term σ, that $\mathrm{Vars}[\sigma] \subseteq \mathcal{V}_s$ and that, for every Δ-equation

$$d(\sigma) \triangleq e,$$

$(\mathrm{Vars}[e] - \mathrm{Vars}[\sigma]) \subseteq \mathcal{V}_r$. Further, assume for expressions e' (not the right-hand side of a Δ-equation) that $\mathrm{Vars}[e'] \subseteq \mathcal{V}_r$.

Next, define a collection of functions, var_v^{σ}, sub_v^{σ}, $\mathrm{scope}_v^{\sigma}$, and abs_v^{σ} that allow us to analyze a σ-instance I. Each variable $y \in \mathrm{Bvars}[\sigma]$ can be used to name a variable var_y^{σ} in I.

Definition 19 (var_y^{σ}) *For an s-term σ and a variable $y \in \mathrm{Bvars}[\sigma]$, define the variable $\mathrm{var}_y^{\sigma}[I]$ by induction on the structure of σ:*

$$\mathrm{var}_y^{\langle\sigma_1,\sigma_2\rangle}[\langle I_1, I_2\rangle] \equiv \mathrm{var}_y^{\sigma_i}[I_i] \quad \textit{(for } y \in \mathrm{Bvars}[\sigma_i] \textit{ where } i \in \{1,2\}\textit{)}$$

$$\mathrm{var}_y^{\lambda z.\sigma_1}[\lambda x.I_1] \equiv \mathrm{var}_y^{\sigma_1}[I_1] \quad (z \neq y)$$

$$\mathrm{var}_y^{\lambda y.\sigma_1}[\lambda x.I_1] \equiv x$$

Each variable $v \in \mathrm{Evars}[\sigma]$ can be used to name a subterm $\mathrm{sub}_v^{\sigma}[I]$ of I.

Definition 20 (sub_v^{σ}) *For all s-terms σ', σ-instances I, and $v \in \mathrm{Evars}[\sigma]$, define the expression $\mathrm{sub}_v^{\sigma}[I]$ by induction on the structure of σ:*

$$\begin{aligned}
&\mathrm{sub}_v^{v}[I] &&\equiv I \\
&\mathrm{sub}_v^{\langle\sigma_1,\sigma_2\rangle}[\langle I_1, I_2\rangle] &&\equiv \mathrm{sub}_v^{\sigma_i}[I_i] \quad (v \in \mathrm{Evars}[\sigma_i]) \\
&\mathrm{sub}_v^{\lambda y.\sigma_1}[\lambda x.I_1] &&\equiv \mathrm{sub}_v^{\sigma_1}[I_1]
\end{aligned}$$

The function $\text{scope}_v^\sigma[I]$ computes the binding environment at of the subterm $\text{sub}_v^\sigma[I]$ in I.

Definition 21 (scope_v^σ) *For all s-terms σ, variables $v \in \text{Evars}[\sigma]$, and σ-instances I, define the vector of variables $\text{scope}_v^\sigma[I]$ by induction on the structure of σ:*

$$\begin{array}{ll} \text{scope}_v^v[I] & \equiv [] \\ \text{scope}_v^{\langle\sigma_1,\sigma_2\rangle}[\langle I_1, I_2\rangle] & \equiv \text{scope}_v^{\sigma_i}[I_i] \\ & \textit{(for } v \in \text{Evars}[\sigma_i] \textit{ where } i = 1 \textit{ or } i = 2\textit{)} \\ \text{scope}_v^{\lambda y.\sigma_1}[\lambda x.I_1] & \equiv x, \text{scope}_v^{\sigma_1}[I_1] \end{array}$$

Since every s-term σ is itself a σ-instance, the expression $\text{scope}_v^\sigma[\sigma]$ is well defined for any $v \in \text{Evars}[\sigma]$.

These definitions can be illustrated with the s-term σ and the σ-instance I given in Equations 3.26 and 3.27:

$$\begin{array}{lcllcl} \text{var}_x^\sigma[I] & \equiv & w, & \text{sub}_a^\sigma[I] & \equiv & \overline{A}, \\ \text{var}_y^\sigma[I] & \equiv & u, & \text{sub}_b^\sigma[I] & \equiv & \overline{B}, \\ & & & \text{sub}_c^\sigma[I] & \equiv & \overline{C}, \\ \text{scope}_a^\sigma[\sigma] & \equiv & [], & \text{scope}_a^\sigma[I] & \equiv & [], \\ \text{scope}_b^\sigma[\sigma] & \equiv & [x], & \text{scope}_b^\sigma[I] & \equiv & [w], \\ \text{scope}_c^\sigma[\sigma] & \equiv & [x, y], & \text{scope}_c^\sigma[I] & \equiv & [w, u]. \end{array}$$

Definition 22 *For an s-term σ, a σ-instance I is said to have a variable* free and bound w.r.t. σ *if* $\text{Fvars}[I] \cap \{\text{var}_y^\sigma[I] \mid y \in \text{Bvars}[\sigma]\} \neq \phi$.

Assume a function $\text{ConvInst}^\sigma[I]$ such that, for any σ-instance I,

1. $I =_\alpha \text{ConvInst}^\sigma[I]$,
2. $\text{ConvInst}^\sigma[I]$ is a σ-instance,
3. $\text{ConvInst}^\sigma[I]$ has no variables free and bound w.r.t. σ,
4. if I has no variables free and bound w.r.t. σ then $\text{ConvInst}^\sigma[I] \equiv I$.

In other words, $\text{ConvInst}^\sigma[I]$ α-converts bound variables in I that are free and bound w.r.t. σ to variables that are not.

Finally, it is possible to define Exp^Δ. Informally, the function $\text{Exp}^\Delta[d(I)]$ first changes the variables of e (the left-hand side of the Δ-equation for $d(\sigma)$) to correspond to those actually used in the σ-instance I by simultaneous α-conversion. It then inserts each subterm of I into the appropriate location in the result by simultaneous capture incurring substitution. However, to ensure that this is well-defined, the instance I must first be converted to $I' = \mathrm{ConvInst}^{\sigma}[I]$.

Definition 23 (Exp^{Δ}) *Given a Δ-equation*

$$d(\sigma) \;\stackrel{\Delta}{=}\; e$$

and a σ-instance I, let

$$\begin{aligned}
\vec{v} &= [v_1, \ldots, v_n] = \mathrm{Evars}[\sigma], \\
\vec{y} &= [y_1, \ldots, y_k] = \mathrm{Bvars}[\sigma], \\
I' &= \mathrm{ConvInst}^{\sigma}[I], \\
\vec{u} &= [\mathrm{var}^{\sigma}_{y_1}[I'], \ldots, \mathrm{var}^{\sigma}_{y_k}[I']], \\
e_i &= \mathrm{sub}^{\sigma}_{v_i}[I'], \\
\vec{w_i} &= \mathrm{scope}^{\sigma}_{v_i}[I'],
\end{aligned}$$

and define $\mathrm{Exp}^{\Delta}[d(I)]$ *as*

$$\mathrm{Exp}^{\Delta}[d(I)] \;\equiv\; \tilde{e}$$

where

$$\tilde{e} \equiv e\left[\frac{\vec{u}}{\vec{y}}\right]^{\alpha}\left[\frac{(\mathrm{sub}^{\sigma}_{v_1}[I'], \{\mathrm{scope}^{\sigma}_{v_1}[I']\})}{v_1} \quad \cdots \quad \frac{(\mathrm{sub}^{\sigma}_{v_n}[I'], \{\mathrm{scope}^{\sigma}_{v_n}[I']\})}{v_n}\right].$$

Returning again to the running example,

$\mathrm{Exp}^{\Delta}[\overline{\forall w \in A, u \in B.C}]$

$$\begin{aligned}
&\equiv \overline{\forall x \in a.\forall y \in b.c}\left[\frac{w}{x}, \frac{u}{y}\right]^{\alpha}\left[\frac{(\overline{A}, \phi)}{a}, \frac{(\overline{B}, \{w\})}{b}, \frac{(\overline{C}, \{w, u\})}{c}\right] \\
&\equiv \overline{\forall w \in a.\forall u \in b.c}\left[\frac{(\overline{A}, \phi)}{a}, \frac{(\overline{B}, \{w\})}{b}, \frac{(\overline{C}, \{w, u\})}{c}\right] \\
&\equiv \forall_1(\langle \overline{A}, \lambda w.\forall_1(\langle \overline{B}, \lambda u.\overline{C}\rangle)\rangle) \\
&\equiv \overline{\forall w \in A.\forall u \in B.C}
\end{aligned}$$

assuming, for sake of a clear presentation, that $\mathrm{ConvInst}[I] \equiv I$, that is

$$\mathrm{Fvars}[\langle \overline{A}, \lambda w.\forall_1(\langle \overline{B}, \lambda u.\overline{C}\rangle)\rangle] \cap \{w, u\} = \phi.$$

A few remarks concerning the function Exp^{Δ} are in order:

1. Exp^{Δ} is an acceptable expansion. See Theorem 12 in the next section for a proof.

2. The expansion $\mathrm{Exp}^{\Delta}[d(I)]$ provides for the preservation of variable names in most instances. Only when a binding variable is free and bound w.r.t. σ in I is it renamed in the expansion $\mathrm{Exp}^{\Delta}[d(I)]$. This situation seems to arise rarely in practice.

3. The expansion $\mathrm{Exp}^{\Delta}[d(I)]$ lends itself well to an efficient implementation (Property 7) in two respects. First, there is no need to construct the expression $\mathcal{F}_e^{\sigma}$ to obtain this expansion. Second, if the types of all variables in $\mathrm{Evars}[\sigma]$ are in T_0 and if e and I are both in normal form, then $\mathrm{Exp}^{\Delta}[d(I)]$ is obtained without reduction and is itself in normal form.

4. There is no danger of violating the assumptions concerning the partition of $\mathcal{V}$ since it is easy to see that $\mathrm{Vars}[\mathrm{Exp}^{\Delta}[d(I)]] \subseteq \mathcal{V}_r$.

4.4 Proof that Exp^{Δ} is an acceptable expansion

This section presents a proof that Exp^{Δ} is an acceptable expansion. First, an expression $\Phi_v^{\sigma}[e]$ is defined in such a way that, for any σ-instance I, the expression $\mathrm{sub}_v^{\sigma}[I]$, with variables of $\mathrm{scope}_v^{\sigma}[I]$ replaced by those of $\mathrm{scope}_v^{\sigma}[\sigma]$, can be obtained by reduction of $\Phi_v^{\sigma}[z][I/z]$.

Definition 24 (abs_v^{σ}) *For all s-terms σ, for all variables $v \in \mathrm{Evars}[\sigma]$, and for all σ-instances I, define abs_v^{σ} by induction on σ:*

$$
\begin{array}{lll}
\mathrm{abs}_v^{v}[I] & \equiv I & \\
\mathrm{abs}_v^{\langle\sigma_1,\sigma_2\rangle}[\langle I_1, I_2\rangle] & \equiv \mathrm{abs}_v^{\sigma_i}[I_i] & (v \in \mathrm{Evars}[\sigma_i],\ i = 1 \ \mathit{or}\ i = 2) \\
\mathrm{abs}_v^{\lambda y.\sigma_1}[\lambda x.I_1] & \equiv \lambda x.\mathrm{abs}_v^{\sigma_1}[I_1] &
\end{array}
$$

Alternatively, abs_v^{σ} could be defined as

Fact 7 $\mathrm{abs}_v^{\sigma}[I] \equiv \lambda \mathrm{scope}_v^{\sigma}[I].\mathrm{sub}_v^{\sigma}[I]$.

The first definition is, of course, easier to work with in proofs that proceed by induction on σ.

Fact 8
Let σ be an s-term and I_1, I_2 be σ-instances. If $I_1 =_{\alpha} I_2$ then then $\mathrm{abs}_v^{\sigma}[I_1] =_{\alpha} \mathrm{abs}_v^{\sigma}[I_2]$.

Fact 9 *Let σ, σ' be an s-terms with $\sigma =_{\alpha} \sigma'$ and let I be a σ-instances, then* $\mathrm{abs}_v^{\sigma}[I] \equiv \mathrm{abs}_v^{\sigma'}[I]$.

Definition 25 (trans_v^σ) *Given an s-term σ, a variable $v \in \mathrm{Evars}[\sigma]$, and any expression $e \in \tau[\sigma]$, define the expression $\mathrm{trans}_v^\sigma[e]$ by induction on the structure of σ:*

$$\begin{aligned}
\mathrm{trans}_v^v[e] &\equiv e \\
\mathrm{trans}_v^{\langle\sigma_1,\sigma_2\rangle}[e] &\equiv \mathrm{trans}_v^{\sigma_i}[\pi_i(e)] \quad (v \in \mathrm{Evars}[\sigma_i],\ i = 1 \text{ or } i = 2) \\
\mathrm{trans}_v^{\lambda y.\sigma_1}[e] &\equiv \lambda y.\mathrm{trans}_v^{\sigma_1}[e(y)]
\end{aligned}$$

Definition 26 (f_v^σ) *Given an s-term σ and a variable $v \in \mathrm{Evars}[\sigma]$, define the expression*

$$f_v^\sigma \ \equiv\ \lambda z^{\tau[\sigma]}.\mathrm{trans}_v^\sigma[z].$$

where z is a new variable such that $z \notin \mathrm{Vars}[\sigma]$.

Again, a definition of f_v^σ could have been given that is somewhat easier to understand but more difficult to work with in inductive proofs:

Fact 10 $f_v^\sigma \equiv \lambda z^{\tau[\sigma]}.\lambda\mathrm{scope}_v^\sigma[\sigma].\Phi_v^\sigma[z]$

Fact 11 *Let e_1 and e_2 be expressions with $e_1 \twoheadrightarrow e_2$, and let σ be an s-term with variable $v \in \mathrm{Evars}[\sigma]$, then $\mathrm{trans}_v^\sigma[e_1] \twoheadrightarrow \mathrm{trans}_v^\sigma[e_2]$.*

The following lemma tells us that, under certain circumstances, substitution commutes with $\mathrm{trans}_v^\sigma[e]$.

Lemma 5 *For all expressions e_1, e_2 and s-terms σ and variables z with*

1. $\mathrm{Fvars}[e_1] \cap \mathrm{Bvars}[\sigma] = \phi$,
2. $z \notin \mathrm{Bvars}[e_1]$ *and* $z \in \mathrm{Fvars}[e_1]$,

for some s-term $\sigma' =_\alpha \sigma$ with $\mathrm{Fvars}[e_2] \cap \mathrm{Bvars}[\sigma'] = \phi$,

$$\mathrm{trans}_v^\sigma[e_1][e_2/z] \equiv \mathrm{trans}_v^{\sigma'}[e_1[e_2/z]].$$

Proof. By induction on the structure of σ.
Case 1: $\sigma \equiv v^\tau$ then $\mathrm{trans}_v^v[e_1][e_2/z] \equiv e_1[e_2/z] \equiv \mathrm{trans}_v^v[e_1[e_2/z]]$ by the definitions of trans_v^σ.
Case 2: $\sigma \equiv \langle\sigma_1,\sigma_2\rangle$ then $v \in Vars[\sigma_1]$ or $v \in Vars[\sigma_2]$. Only the former case will be considered because the latter can be treated symmetrically. Since I is a σ-instance, $I \equiv \langle I_1, I_2\rangle$ for some σ_1-instance I_1 and some σ_2-instance I_2. Then

$$\begin{aligned}
\mathrm{trans}_v^{\langle\sigma_1,\sigma_2\rangle}[e_1][e_2/z] &\equiv \mathrm{trans}_v^{\sigma_1}[\pi_1(e_1)][e_2/z] \\
&\equiv \mathrm{trans}_v^{\sigma_1'}[\pi_1(e_1)[e_2/z]] && \text{(by induction)} \\
&\equiv \mathrm{trans}_v^{\sigma_1'}[\pi_1(e_1[e_2/z])] \\
&\equiv \mathrm{trans}_v^{\langle\sigma_1',\sigma_2'\rangle}[e_1[e_2/z] && \text{(for any } \sigma_2' =_\alpha \sigma_2)
\end{aligned}$$

Case 3: $\sigma \equiv \lambda y.\sigma_1$ and $I \equiv \lambda x.I_1$ for some σ_1-instance I_1. Then by the definition of trans_v^σ

$$\mathrm{trans}_v^{\lambda y.\sigma_1}[e_1][e_2/z] \equiv (\lambda y.\mathrm{trans}_v^{\sigma_1}[e_1(y)])[e_2/z],$$

which leaves three cases to consider:
Case 3.1: $y = z$. This cannot occur by condition 2.
Case 3.2: $y \notin \mathrm{Fvars}[e_2]$. Then

$$\begin{array}{ll} \equiv \lambda y.\mathrm{trans}_v^{\sigma_1}[e_1(y)][e_2/z] & \\ \equiv \lambda y.\mathrm{trans}_v^{\sigma_1'}[(e_1(y))[e_2/z]] & \text{(by induction)} \\ \equiv \lambda y.\mathrm{trans}_v^{\sigma_1'}[e_1[e_2/z](y)] & \text{(since } y \neq z \text{ by condition 2)} \\ \equiv \mathrm{trans}_v^{\lambda y.\sigma_1'}[e_1[e_2/z]] & \end{array}$$

Case 3.3: $y \in \mathrm{Fvars}[e_2]$. Then let w be a new variable so that

$$\begin{array}{ll} \equiv \lambda w.\mathrm{trans}_v^{\sigma_1}[e_1(y)][w/y][e_2/z] & \\ \equiv \lambda w.\mathrm{trans}_v^{\sigma_1}[e_1(w)][e_2/z] & \\ \equiv \lambda w.\mathrm{trans}_v^{\sigma_1'}[(e_1(w))[e_2/z]] & \text{(by induction)} \\ \equiv \lambda w.\mathrm{trans}_v^{\sigma_1'}[e_1[e_2/z](w)] & \\ \quad \text{(since } w \neq z \text{ by condition choice of } w\text{)} & \\ \equiv \mathrm{trans}_v^{\lambda w.\sigma_1'}[e_1[e_2/z]] & \end{array}$$

□

The following lemma tells us that $\mathrm{trans}_v^\sigma[I]$ does indeed extract the expression $\mathrm{abs}_v^\sigma[I]$ from a σ-instance I by reduction.

Lemma 6 *If σ is an s-term with $v \in \mathrm{Evars}[\sigma]$ and I is a σ-instance with* $\mathrm{Fvars}[I] \cap \mathrm{Bvars}[\sigma] = \phi$ *then* $\mathrm{trans}_v^\sigma[I] \twoheadrightarrow \mathrm{abs}_v^\sigma[I]$.

Proof. By induction on the structure of σ.
Case 1: $\sigma \equiv v^\tau$. Then $\mathrm{trans}_v^v[I] \equiv I \equiv \mathrm{abs}_v^v[I]$ by the definitions of trans_v^σ and abs_v^σ.
Case 2: $\sigma \equiv \langle\sigma_1, \sigma_2\rangle$. Then $v \in Vars[\sigma_1]$ or $v \in Vars[\sigma_2]$. Only the former case will be considered because the latter can be treated symmetrically. Since I is a σ-instance, $I \equiv \langle I_1, I_2\rangle$ for some σ_1-instance I_1 and some σ_2-instance I_2,

$$\begin{array}{rcll} \mathrm{trans}_v^\sigma[I] & \equiv & \mathrm{trans}_v^{\langle\sigma_1,\sigma_2\rangle}[\langle I_1, I_2\rangle] & \\ & \equiv & \mathrm{trans}_v^{\sigma_1}[\pi_1(\langle I_1, I_2\rangle)] & \\ & \longrightarrow & \mathrm{trans}_v^{\sigma_1}[I_1] & \text{(by Fact 11)} \\ & \twoheadrightarrow & \mathrm{abs}_v^{\sigma_1}[I_1] & \text{(by induction)} \\ & \equiv & \mathrm{abs}_v^{\langle\sigma_1,\sigma_2\rangle}[\langle I_1, I_2\rangle] & \\ & \equiv & \mathrm{abs}_v^\sigma[I]. & \end{array}$$

Case 3: $\sigma \equiv \lambda y.\sigma_1$. Then $I \equiv \lambda x.I_1$ for some σ_1-instance I_1 and

$$
\begin{array}{rll}
\mathrm{trans}_v^{\sigma}[I] & \equiv \mathrm{trans}_v^{\lambda y.\sigma_1}[\lambda x.I_1] & \\
& \equiv \lambda y.\mathrm{trans}_v^{\sigma_1}[(\lambda x.I_1)(y)] & \\
& \longrightarrow \lambda y.\mathrm{trans}_v^{\sigma_1}[I_1[y/x]] & \text{(by Fact 11)} \\
& \twoheadrightarrow \lambda y.\mathrm{abs}_v^{\sigma_1}[I_1[y/x]] & \text{(by induction)} \\
& \equiv \mathrm{abs}_v^{\lambda y.\sigma_1}[\lambda y.I_1[y/x]] & \\
& =_{\alpha} \mathrm{abs}_v^{\lambda y.\sigma_1}[\lambda x.I_1[y/x][x/y]] & \text{(by Fact 9)} \\
& \equiv \mathrm{abs}_v^{\lambda y.\sigma_1}[\lambda x.I_1] & (\mathrm{Fvars}[I] \cap \mathrm{Bvars}[\sigma] = \phi) \\
& \equiv absv\sigma[I]. &
\end{array}
$$

□

The expression $\mathrm{abs}_v^{\sigma}[I]$ can be obtained (modulo α-conversion) by reduction of the expression $f_v^{\sigma}(I)$.

Lemma 7 *If σ is an s-term with $v \in \mathrm{Evars}[\sigma]$ and I is a σ-instance then*

$$f_v^{\sigma}(I) \twoheadrightarrow \mathrm{abs}_v^{\sigma}[I].$$

Proof.

$$
\begin{array}{rll}
f_v^{\sigma}(I) \equiv & (\lambda z.\mathrm{trans}_v^{\sigma}[z])(I) & \\
\longrightarrow & \mathrm{trans}_v^{\sigma}[z][I/z] & \\
\equiv & \mathrm{trans}_v^{\sigma'}[I] & \text{(by Lemma 5)} \\
\twoheadrightarrow & \mathrm{abs}_v^{\sigma'}[I] & \text{(by Lemma 6)} \\
\equiv & \mathrm{abs}_v^{\sigma}[I] & \text{(by Fact 9)}
\end{array}
$$

□

This can be illustrated by using σ and I of Equations 3.26 and 3.27

$$
\begin{array}{rclrcl}
\mathrm{abs}_a^{\sigma}[I] & \equiv & \overline{A}, & f_a^{\sigma}(I) & \twoheadrightarrow & \overline{A}, \\
\mathrm{abs}_b^{\sigma}[I] & \equiv & \lambda w.\overline{B}, & f_b^{\sigma}(I) & \twoheadrightarrow & \lambda x.\overline{B}[x/w], \\
\mathrm{abs}_c^{\sigma}[I] & \equiv & \lambda w.\lambda u.\overline{C} & f_c^{\sigma}(I) & \twoheadrightarrow & \lambda x.\lambda y.\overline{C}[x/w][y/u].
\end{array}
$$

The following two lemmas are of no general interest, but allows us to obtain Corollary 10, which, in turn, justifies one important step in the proof of Theorem 12.

Lemma 8 *Given an expression e, an $n \geq 1$, a set $\{v_1, \ldots, v_n\}$ of pairwise distinct variables, a collection of variable vectors $\vec{y_1}, \ldots, \vec{y_n}$ (each $\vec{y_i}$ possibly null), and any variable z with*

1. $z \notin \{\vec{v}\}$,

2. $x \notin \{\vec{v}\} \cup (\bigcup_{1 \leq i \leq n} \{\vec{y_i}\})$,

3. $\text{Bvars}[e] \cap \{\vec{v}\} = \phi$,

4. $z \notin \text{Bvars}[e]$,

$$e\left\{ \frac{v_1(\vec{y_1})}{v_1} \;\cdots\; \frac{v_n(\vec{y_n})}{v_n} \right\}\left[\frac{z}{x} \right] \;\equiv\; e\left[\frac{z}{x} \right]\left\{ \frac{v_1(\vec{y_1})}{v_1} \;\cdots\; \frac{v_n(\vec{y_n})}{v_n} \right\}.$$

Proof. By induction on the structure of e.
Case 1: The result is trivial if e is a constant.
Case 2: If e is a variable, $e \equiv v$, there are two cases to consider.
Case 2.1: $v \notin \{\vec{v}\}$. Then

$$\begin{aligned}
& v\left\{ \frac{v_1(\vec{y_1})}{v_1} \;\cdots\; \frac{v_n(\vec{y_n})}{v_n} \right\}\left[\frac{z}{x} \right] \\
\equiv \quad & v\left[\frac{z}{x} \right] \\
\equiv \quad & v\left[\frac{z}{x} \right]\left\{ \frac{v_1(\vec{y_1})}{v_1} \;\cdots\; \frac{v_n(\vec{y_n})}{v_n} \right\}
\end{aligned}$$

(since $z \notin \bigcup_{1 \leq i \leq n} \text{Fvars}[v_i(\vec{y_i})]$ by conditions 1 and 2.)

Case 2.2: $v = v_j$ for some $j \in \{1, \ldots, n\}$. Then

$$\begin{aligned}
& v_j\left\{ \frac{v_1(\vec{y_1})}{v_1} \;\cdots\; \frac{v_n(\vec{y_n})}{v_n} \right\}\left[\frac{z}{x} \right] \\
\equiv \quad & (v_j(\vec{y_j}))\left[\frac{z}{x} \right] \\
\equiv \quad & (v_j(\vec{y_j})) \quad (\text{since } x \notin \{\vec{v}\} \cup \{\vec{y}\}) \\
\equiv \quad & v_j\left\{ \frac{v_1(\vec{y_1})}{v_1} \;\cdots\; \frac{v_n(\vec{y_n})}{v_n} \right\} \quad (\text{since } x \notin \{\vec{v}\}) \\
\equiv \quad & v_j\left[\frac{z}{x} \right]\left\{ \frac{v_1(\vec{y_1})}{v_1} \;\cdots\; \frac{v_n(\vec{y_n})}{v_n} \right\}
\end{aligned}$$

Cases 3, 4, and 5: e is of the form $\pi_i(e')$, $\langle e'', e''\rangle$, or $e'(e'')$. The results go through easily since substitution is defined by a simple induction in these cases and conditions 1–4 still hold.
Case 6: $e \equiv \lambda w.e'$. Following the last clause in the definition of substitution, there is only the case of $x \notin \vec{v}$ to consider (since $w \in \vec{v}$. by condition 3), so that

$$(\lambda w.e')\left\{ \frac{v_1(\vec{y_1})}{v_1} \;\cdots\; \frac{v_n(\vec{y_n})}{v_n} \right\}\left[\frac{z}{x} \right]$$

$$\equiv \quad (\lambda w.e'\left\{ \frac{v_1(\vec{y_1})}{v_1} \;\cdots\; \frac{v_n(\vec{y_n})}{v_n} \right\})\left[\frac{z}{x} \right]$$

at which point there are three cases to consider (again since $w \notin \{\vec{v}\}$):
Case 6.1 : $w = x$. Then

$$\equiv \quad (\lambda w.e'\left\{ \frac{v_1(\vec{y_1})}{v_1} \;\cdots\; \frac{v_n(\vec{y_n})}{v_n} \right\})\left[\frac{z}{x} \right]$$

$$\equiv \quad (\lambda w.e'\left\{ \frac{v_1(\vec{y_1})}{v_1} \;\cdots\; \frac{v_n(\vec{y_n})}{v_n} \right\})$$

$$\equiv \quad (\lambda w.e')\left\{ \frac{v_1(\vec{y_1})}{v_1} \;\cdots\; \frac{v_n(\vec{y_n})}{v_n} \right\}$$

$$\equiv \quad (\lambda w.e')\left[\frac{z}{x} \right]\left\{ \frac{v_1(\vec{y_1})}{v_1} \;\cdots\; \frac{v_n(\vec{y_n})}{v_n} \right\}$$

Case 6.2 : $w \neq x$ and $w \neq z$. Then

$$\equiv \quad (\lambda w.e'\left\{ \frac{v_1(\vec{y_1})}{v_1} \;\cdots\; \frac{v_n(\vec{y_n})}{v_n} \right\})\left[\frac{z}{x} \right]$$

$$\equiv \quad (\lambda w.e'\left\{ \frac{v_1(\vec{y_1})}{v_1} \;\cdots\; \frac{v_n(\vec{y_n})}{v_n} \right\}\left[\frac{z}{x} \right])$$

$$\equiv \quad (\lambda w.e'\left[\frac{z}{x} \right]\left\{ \frac{v_1(\vec{y_1})}{v_1} \;\cdots\; \frac{v_n(\vec{y_n})}{v_n} \right\})$$

(by induction since conditions 1-4 still hold)

$$\equiv \quad (\lambda w.e')\left[\frac{z}{x} \right]\left\{ \frac{v_1(\vec{y_1})}{v_1} \;\cdots\; \frac{v_n(\vec{y_n})}{v_n} \right\}$$

Case 6.3: $w \neq x$ and $w = z$. This cannot occur by condition 4. □

Lemma 9 *For any expression e and any $k \geq 1$ pairwise distinct variables*

$$\vec{y} \equiv [y_1, y_2, \ldots y_k]$$

and any function s mapping variables to variables, let

$$\vec{u} \equiv [u_1, u_2, \ldots u_k]$$

where $u_i = s(y_i)$. Then if

1. $\{\vec{y}\} \cap \{\vec{u}\} = \{\vec{y}\} \cap \{\vec{v}\} = \{\vec{v}\} \cap \{\vec{u}\} = \phi$,

2. $\mathrm{Bvars}[e_i] = \phi$,

3. $\mathrm{Vars}[e_i]$ $(= \mathrm{Evars}[e_i])$ $= \{\vec{y_i}\} \cup \{v_i\}$, *where* $\{\vec{y_i}\} \subseteq \{\vec{y}\}$,

$$e\left\{ \frac{e_1}{v_1} \cdots \frac{e_n}{v_n} \right\}\left[\frac{\vec{u}}{\vec{y}}\right]^\alpha\left[\frac{\vec{u}}{\vec{y}}\right] =_\alpha$$

$$e\left[\frac{\vec{u}}{\vec{y}}\right]^\alpha\left[\frac{\vec{u}}{\vec{y}}\right]\left\{ \frac{e_1[\vec{u}/\vec{y}]}{v_1} \cdots \frac{e_n[\vec{u}/\vec{y}]}{v_n} \right\}.$$

Proof. By induction on the structure of e: Case 1: The result is trivial if e is a constant.
Case 2: e is a variable, $e \equiv x$, which involves three subcases to consider.
Case2.1: $x \in \vec{v}$, say $x \equiv v_j$. Then both sides of the equation evaluate to $e_j[\vec{u}/\vec{y}]$, since $\vec{y} \cap \vec{v} = \phi$ and $\mathrm{Bvars}[e_j] = \phi$, it is easy to see from the definition of simultaneous α-conversion that $e_j[\vec{u}/\vec{y}]^\alpha \equiv e_j$.
Case 2.2: $x \in \vec{y}$, say $x \equiv y_j$. Then both sides of the equation evaluate to u_j since $\vec{y} \cap \vec{u} \cap \vec{v} = \phi$.
Case 2.3: $x \notin \vec{y} \cup \vec{u}$. Then both sides of the equation evaluate to x.
Cases 3, 4, and 5: e is of the form $\pi_i(e')$, $\langle e'', e''\rangle$, or $e'(e'')$. The result goes through easily since substitution and simultaneous α-conversion are defined by induction and since the conditions still hold.
Case 6: $e \equiv \lambda x.e_0$. There are two subcases to consider.
Case 6.1: $x \equiv y_j$. Evaluating the left side of the equation,

$$
\begin{aligned}
&(\lambda y_j.e_0)\left\{ \frac{e_1}{v_1} \cdots \frac{e_n}{v_n} \right\}[\vec{u}/\vec{y}]^\alpha[\vec{u}/\vec{y}] \\
&\equiv \quad (\lambda y_j.e_0\left\{ \frac{e_1}{v_1} \cdots \frac{e_n}{v_n} \right\})[\vec{u}/\vec{y}]^\alpha[\vec{u}/\vec{y}] \\
&\equiv \quad (\lambda u_j.e_0\left\{ \frac{e_1}{v_1} \cdots \frac{e_n}{v_n} \right\}[\vec{u}/\vec{y}]^\alpha[u_j/y_j])[\vec{u}/\vec{y}] \\
&=_\alpha \quad \lambda u_j.e_0\left\{ \frac{e_1}{v_1} \cdots \frac{e_n}{v_n} \right\}[\vec{u}/\vec{y}]^\alpha[u_j/y_j][\vec{u}/\vec{y}]/\{y_j\} \\
&\text{(by Fact 5)} \\
&=_\alpha \quad \lambda u_j.e_0\left\{ \frac{e_1}{v_1} \cdots \frac{e_n}{v_n} \right\}[\vec{u}/\vec{y}]^\alpha[\vec{u}/\vec{y}] \\
&\text{(by Fact 6).}
\end{aligned}
$$

Now, turning to the right side of the equation,

$$(\lambda y_j.e_0)[\vec{u}/\vec{y}]^\alpha[\vec{u}/\vec{y}]\left\{ \frac{e_1[\vec{u}/\vec{y}]}{v_1} \cdots \frac{e_n[\vec{u}/\vec{y}]}{v_n} \right\}$$

$$
\begin{aligned}
&\equiv \quad (\lambda u_j.e_0[\vec{u}/\vec{y}]^{\alpha}[u_j/y_j])[\vec{u}/\vec{y}]\left\{ \frac{e_1[\vec{u}/\vec{y}]}{v_1} \cdots \frac{e_n[\vec{u}/\vec{y}]}{v_n} \right\} \\
&=_\alpha \quad (\lambda u_j.e_0[\vec{u}/\vec{y}]^{\alpha}[u_j/y_j][\vec{u}/\vec{y}]/\{y_j\})\left\{ \frac{e_1[\vec{u}/\vec{y}]}{v_1} \cdots \frac{e_n[\vec{u}/\vec{y}]}{v_n} \right\}
\end{aligned}
$$

(by Fact 5)

$$
=_\alpha \quad (\lambda u_j.e_0[\vec{u}/\vec{y}]^{\alpha}[\vec{u}/\vec{y}])\left\{ \frac{e_1[\vec{u}/\vec{y}]}{v_1} \cdots \frac{e_n[\vec{u}/\vec{y}]}{v_n} \right\}
$$

(by Fact 6)

$$
\equiv \quad \lambda u_j.e_0[\vec{u}/\vec{y}]^{\alpha}[\vec{u}/\vec{y}]\left\{ \frac{e_1[\vec{u}/\vec{y}]}{v_1} \cdots \frac{e_n[\vec{u}/\vec{y}]}{v_n} \right\}.
$$

which is $=_\alpha$ to the result on the left by induction.
Case 6.2: $x \equiv u_j$. Evaluating the left side of the equation,

$$
\begin{aligned}
&(\lambda u_j.e_0)\left\{ \frac{e_1}{v_1} \cdots \frac{e_n}{v_n} \right\}[\vec{u}/\vec{y}]^{\alpha}[\vec{u}/\vec{y}] \\
&\quad\equiv \quad (\lambda u_j.e_0\left\{ \frac{e_1}{v_1} \cdots \frac{e_n}{v_n} \right\})[\vec{u}/\vec{y}]^{\alpha}[\vec{u}/\vec{y}] \\
&\quad\equiv \quad (\lambda z_1.e_0\left\{ \frac{e_1}{v_1} \cdots \frac{e_n}{v_n} \right\}[z_1/u_j][\vec{u}/\vec{y}]^{\alpha})[\vec{u}/\vec{y}] \\
&\quad\equiv \quad (\lambda z_1.e_0[z_1/u_j]\left\{ \frac{e_1}{v_1} \cdots \frac{e_n}{v_n} \right\}[\vec{u}/\vec{y}]^{\alpha})[\vec{u}/\vec{y}]
\end{aligned}
$$

since $\vec{u} \cap \text{Vars}[e_i] = \phi$

$$
\equiv \quad \lambda z_1.(e_0[z_1/u_j])\left\{ \frac{e_1}{v_1} \cdots \frac{e_n}{v_n} \right\}[\vec{u}/\vec{y}]^{\alpha}[\vec{u}/\vec{y}]
$$

where $z_1 = \text{NewVar}[\{\vec{u}\} \cup \{\vec{y}\} \cup \text{Vars}[e_0\left\{ \frac{e_1}{v_1} \cdots \frac{e_n}{v_n} \right\}]]$. Now, turning to the right side of the equation,

$$
\begin{aligned}
&(\lambda u_j.e_0)\left[\frac{\vec{u}}{\vec{y}} \right]^{\alpha}[\vec{u}/\vec{y}]\left\{ \frac{e_1[\vec{u}/\vec{y}]}{v_1} \cdots \frac{e_n[\vec{u}/\vec{y}]}{v_n} \right\} \\
&\quad\equiv \quad (\lambda z_2.e_0\left[\frac{z_2}{u_j} \right][\vec{u}/\vec{y}]^{\alpha})[u_j/y_j])[\vec{u}/\vec{y}]\left\{ \frac{e_1[\vec{u}/\vec{y}]}{v_1} \cdots \frac{e_n[\vec{u}/\vec{y}]}{v_n} \right\} \\
&\quad\equiv \quad \lambda z_2.(e_0[z_2/u_j])[\vec{u}/\vec{y}]^{\alpha}[u_j/y_j])[\vec{u}/\vec{y}]\left\{ \frac{e_1[\vec{u}/\vec{y}]}{v_1} \cdots \frac{e_n[\vec{u}/\vec{y}]}{v_n} \right\}.
\end{aligned}
$$

where $z_2 = \text{NewVar}[\{\vec{u}\} \cup \{\vec{y}\} \cup \text{Vars}[e_0]]$. But $z_1 = z_2$ since $\text{Vars}[e_i] \subseteq (\vec{v} \cup \vec{y})$ by condition 3, so that the left- and right-hand sides of the equation are $=_\alpha$ by induction. □.

Corollary 10 *For any expression e and any $k \geq 1$ pairwise distinct variables*

$$
\vec{y} \equiv [y_1, y_2, \ldots y_k]
$$

and any function s mapping variables to variables, let

$$\vec{u} \;\equiv\; [u_1, u_2, \ldots u_k]$$

where $u_i = s(y_i)$. *Then if*

1. $\{\vec{y}\} \cap \{\vec{u}\} = \phi$,
2. $\{\vec{y_i}\} \subseteq \{\vec{y}\}$,
3. *every occurrence of* v_i *in* e *is in the scope of each variable in* $\vec{y_i}$,

$$e\left\{\begin{array}{ccc}\dfrac{v_1(\vec{y_1})}{v_1} & \cdots & \dfrac{v_n(\vec{y_n})}{v_n}\end{array}\right\}\left[\begin{array}{c}\dfrac{\vec{u}}{\vec{y}}\end{array}\right]^{\alpha} =_\alpha e\left[\begin{array}{c}\dfrac{\vec{u}}{\vec{y}}\end{array}\right]^{\alpha}\left\{\begin{array}{ccc}\dfrac{v_1(\vec{w_1})}{v_1} & \cdots & \dfrac{v_n(\vec{w_n})}{v_n}\end{array}\right\}$$

where $\vec{w_i} = s(\vec{y_i})$.

The proof that Exp^{Δ} is acceptable will depend on the following rather technical result. Recall that $\mathcal{V} = \mathcal{V}_s \cup \mathcal{V}_r$, where $\mathcal{V}_s \cap \mathcal{V}_r = \phi$.

Lemma 11 *Given an expression* e, *an* $n \geq 1$, *a set* $\{\vec{v}\} = \{v_1, \ldots, v_n\}$ *of pairwise distinct variables, and any collection of variable vectors* $\vec{y_1}, \ldots, \vec{y_n}$ *(each* $\vec{y_i}$ *possibly null) with*

1. $\{\vec{y_i}\} \subseteq \{\vec{y}\}$ *for* $i \in \{1, \ldots, n\}$,
2. $\{\vec{y}\} \cap (\bigcup_{1 \leq i \leq n} \mathrm{Fvars}[\lambda\vec{y_i}.e_i]) = \phi$,
3. $\mathrm{Bvars}[e] \cap \{\vec{v}\} = \phi$,
4. $\{\vec{v}\} \subset \mathcal{V}_s$,

$$e\left\{\begin{array}{ccc}\dfrac{v_1(\vec{y_1})}{v_1} & \cdots & \dfrac{v_n(\vec{y_n})}{v_n}\end{array}\right\}\left[\begin{array}{ccc}\dfrac{\lambda\vec{y_1}.e_1}{v_1} & \cdots & \dfrac{\lambda\vec{y_n}.e_n}{v_n}\end{array}\right]$$

$$\twoheadrightarrow \; e\left[\begin{array}{ccc}\dfrac{(e_1, \{\vec{y_1}\})}{v_1} & \cdots & \dfrac{(e_n, \{\vec{y_n}\})}{v_n}\end{array}\right]$$

Proof. By induction on the structure of e.
Case 1: The result is trivial if e is a constant.
Case 2: e is a variable, $e \equiv v$, which involves two subcases to consider. Case2.1: $v \notin \{\vec{v}\}$. Trivial. Case 2.2: $v = v_j$ for some $j \in \{1, \ldots, n\}$. Then

$$v_j \left\{ \frac{v_1(\vec{y_1})}{v_1} \cdots \frac{v_n(\vec{y_n})}{v_n} \right\} \left[\frac{\lambda\vec{y_1}.e_1}{v_1} \cdots \frac{\lambda\vec{y_n}.e_n}{v_n} \right]$$

$$\begin{array}{ll} \equiv & (\lambda\vec{y_j}.e_j)(\vec{y_j}) \\ \twoheadrightarrow & e_i \\ \equiv & v_j \left[\dfrac{(e_1, \{\vec{y_1}\})}{v_1} \cdots \dfrac{(e_n, \{\vec{y_n}\})}{v_n} \right] \end{array}$$

Cases 3, 4, and 5: e is of the form $\pi_i(e')$, $\langle e'', e''\rangle$, or $e'(e'')$. The results go through easily since substitution is defined by a simple induction in these cases and conditions $1-4$ still hold.

Case 6: $e \equiv \lambda x.e'$. Following the last clause in the definition of substitution, there is only the case of $x \notin \vec{v}$ to consider (since $x \in \vec{v}$. by condition 3), so that

$$(\lambda x.e')\left\{ \frac{v_1(\vec{y_1})}{v_1} \cdots \frac{v_n(\vec{y_n})}{v_n} \right\} \left[\frac{\lambda\vec{y_1}.e_1}{v_1} \cdots \frac{\lambda\vec{y_n}.e_n}{v_n} \right]$$

$$\equiv \quad (\lambda x.e'\left\{ \frac{v_1(\vec{y_1})}{v_1} \cdots \frac{v_n(\vec{y_n})}{v_n} \right\}) \left[\frac{\lambda\vec{y_1}.e_1}{v_1} \cdots \frac{\lambda\vec{y_n}.e_n}{v_n} \right]$$

at which point there are two cases to consider (again since $x \notin \{\vec{v}\}$):

Case 6.1 : $x \notin \bigcup_{1\le i\le n} \text{Fvars}[\lambda\vec{y_i}.e']$.

$$\begin{array}{ll} \equiv & (\lambda x.e'\left\{ \dfrac{v_1(\vec{y_1})}{v_1} \cdots \dfrac{v_n(\vec{y_n})}{v_n} \right\} \left[\dfrac{\lambda\vec{y_1}.e_1}{v_1} \cdots \dfrac{\lambda\vec{y_n}.e_n}{v_n} \right]) \\ \twoheadrightarrow & (\lambda x.e' \left[\dfrac{(e_1, \{\vec{y_1}\})}{v_1} \cdots \dfrac{(e_n, \{\vec{y_n}\})}{v_n} \right]) \quad \text{(by induction)} \\ \equiv & (\lambda x.e') \left[\dfrac{(e_1, \{\vec{y_1}\})}{v_1} \cdots \dfrac{(e_n, \{\vec{y_n}\})}{v_n} \right] \end{array}$$

since $x \notin \bigcup_{1\le i\le n} \text{Fvars}[\lambda\vec{y_i}.e_1] = \bigcup_{1\le i\le n}(\text{Fvars}[e_1] - \{\vec{y_i}\})$

Case 6.2 :$x \in \bigcup_{1\le i\le n} \text{Fvars}[\lambda\vec{y_i}.e_1]$. Looking at the expression

$$(\lambda x.e'\left\{ \frac{v_1(\vec{y_1})}{v_1} \cdots \frac{v_n(\vec{y_n})}{v_n} \right\}) \left[\frac{\lambda\vec{y_1}.e_1}{v_1} \cdots \frac{\lambda\vec{y_n}.e_n}{v_n} \right]$$

there are three subcases to consider.

Case 6.2.1 : $x \in \{\vec{y}\}$: This cannot occur by condition 2.

Case 6.2.2 : $x \notin \{\vec{y}\}$ and $z \in \{\vec{v}\}$. This cannot occur by condition 4.

Case 6.2.3 :$x \notin \{\vec{y}\}$ and $z \notin \{\vec{v}\}$.

$$(\lambda x.e'\left\{\frac{v_1(\vec{y_1})}{v_1}\ \cdots\ \frac{v_n(\vec{y_n})}{v_n}\right\})\left[\frac{\lambda\vec{y_1}.e_1}{v_1}\ \cdots\ \frac{\lambda\vec{y_n}.e_n}{v_n}\right]$$

$$\equiv\quad \lambda z.e'\left\{\frac{v_1(\vec{y_1})}{v_1}\ \cdots\ \frac{v_n(\vec{y_n})}{v_n}\right\}\left[\frac{z}{x}\right]\left[\frac{\lambda\vec{y_1}.e_1}{v_1}\ \cdots\ \frac{\lambda\vec{y_n}.e_n}{v_n}\right]$$

$$\equiv\quad \lambda z.e'\left[\frac{z}{x}\right]\left\{\frac{v_1(\vec{y_1})}{v_1}\ \cdots\ \frac{v_n(\vec{y_n})}{v_n}\right\}\left[\frac{\lambda\vec{y_1}.e_1}{v_1}\ \cdots\ \frac{\lambda\vec{y_n}.e_n}{v_n}\right]$$

by Lemma 8 since $z \notin \{\vec{v}\}$ by $\{\vec{v}\} \subset \mathcal{V}_s$

and $z \notin \text{Bvars}[e]$ by the definition of substitution.

$$\twoheadrightarrow\quad (\lambda z.e'\left[\frac{z}{x}\right]\left[\frac{(e_1,\{\vec{y_1}\})}{v_1}\ \cdots\ \frac{(e_n,\{\vec{y_n}\})}{v_n}\right])$$

by induction (condition 3 still holds since $\{\vec{v}\}$ are special)

$$\equiv\quad (\lambda z.e'\left[\frac{z}{x}\right])\left[\frac{(e_1,\{\vec{y_1}\})}{v_1}\ \cdots\ \frac{(e_n,\{\vec{y_n}\})}{v_n}\right]$$

since $z \notin \bigcup_{1\le i\le n}\text{Fvars}[\lambda\vec{y_i}.e_1] = \bigcup_{1\le i\le n}(\text{Fvars}[e_1] - \{\vec{y_i}\})$

$$=_\alpha\quad (\lambda x.e')\left[\frac{(e_1,\{\vec{y_1}\})}{v_1}\ \cdots\ \frac{(e_n,\{\vec{y_n}\})}{v_n}\right]$$

□

Finally, an expression equivalent to $\mathcal{F}^\sigma_e$ is defined:

Definition 27 ($\hat{\mathcal{F}}^\sigma_e$) *Let σ be an s-term with* $\text{Evars}[\sigma] = \{v_1 \ldots v_n\}$ *and let e be any expression. Let* $z = \text{NewVar}[\text{Vars}[\sigma] \cup \text{Vars}[e]]$. *Define the expression*

$$\hat{\mathcal{F}}^\sigma_e \;\equiv\; \lambda z^{\tau[\sigma]}.e\left\{\frac{f^\sigma_{v_1}(z)(\vec{y_1})}{v_1}\ \cdots\ \frac{f^\sigma_{v_n}(z)(\vec{y_n})}{v_n}\right\}$$

where $\vec{y_i} \equiv scope^\sigma_{v_i}[\sigma]$.

Fact 12 $\hat{\mathcal{F}}^\sigma_e =_\Lambda \mathcal{F}^\sigma_e$.

The main theorem can now be proved.

Theorem 12 Exp^Δ *is an acceptable expansion.*

Proof. Assume that

$$d(\sigma) \triangleq e$$

is a well-formed Δ-equation and that I is a σ-instance. Let $I' = \mathrm{ConvInst}[I]$ and let

$$\begin{aligned} \vec{y_i} &\equiv \mathrm{scope}^{\sigma}_{v_i}[\sigma] \\ \mathrm{abs}^{\sigma}_{v_i}[I'] &\equiv \lambda \mathrm{scope}^{\sigma}_{v}[I'].\mathrm{sub}^{\sigma}_{v_i}[I'] \equiv \lambda \vec{w_i}.\mathrm{sub}^{\sigma}_{v_i}[I']. \end{aligned}$$

Figure 4.3 presents the proof that Exp^{Δ} is an acceptable expansion. $\square$

$$d(I) =_\alpha d(I')$$

$$\longrightarrow \mathcal{F}^\sigma_e(I')$$

$$=_\Lambda \hat{\mathcal{F}}^\sigma_e(I')$$

(by Fact 12)

$$\equiv \ (\lambda z.e\left\{ \frac{f^\sigma_{v_1}(z)(\vec{y_1})}{v_1} \ \cdots \ \frac{f^\sigma_{v_n}(z)(\vec{y_n})}{v_n} \right\})(I')$$

$$\equiv \ (\lambda z.e\left\{ \frac{v_1(\vec{y_1})}{v_1} \ \cdots \ \frac{v_n(\vec{y_n})}{v_n} \right\}\left[\frac{f^\sigma_{v_1}(z)}{v_1} \ \cdots \ \frac{f^\sigma_{v_n}(z)}{v_n} \right])(I')$$

$$\longrightarrow \ e\left\{ \frac{v_1(\vec{y_1})}{v_1} \ \cdots \ \frac{v_n(\vec{y_n})}{v_n} \right\}\left[\frac{f^\sigma_{v_1}(I')}{v_1} \ \cdots \ \frac{f^\sigma_{v_n}(I')}{v_n} \right]$$

$$\twoheadrightarrow \ e\left\{ \frac{v_1(\vec{y_1})}{v_1} \ \cdots \ \frac{v_n(\vec{y_n})}{v_n} \right\}\left[\frac{\mathrm{abs}^\sigma_{v_1}(I')}{v_1} \ \cdots \ \frac{\mathrm{abs}^\sigma_{v_n}(I')}{v_n} \right]$$

(By Lemmas 7 and Fact 3)

$$=_\alpha \ e\left\{ \frac{v_1(\vec{y_1})}{v_1} \ \cdots \ \frac{v_n(\vec{y_n})}{v_n} \right\}\left[\frac{\vec{u}}{\vec{y}} \right]^\alpha \left[\frac{\mathrm{abs}^\sigma_{v_1}(I')}{v_1} \ \cdots \ \frac{\mathrm{abs}^\sigma_{v_n}(I')}{v_n} \right]$$

(by Fact 4 and Lemma 4 and $\mathrm{Vars}[\sigma] \subset \mathcal{V}_s$)

$$\equiv \ e\left[\frac{\vec{u}}{\vec{y}} \right]^\alpha \left\{ \frac{v_1(\vec{w_1})}{v_1} \ \cdots \ \frac{v_n(\vec{w_n})}{v_n} \right\}\left[\frac{e_1}{v_1} \ \cdots \ \frac{e_n}{v_n} \right]$$

(where $e_i \equiv \lambda\vec{w_i}.\mathrm{sub}^\sigma_{v_i}[I']$, by Corollary 10)

$$\twoheadrightarrow \ e\left[\frac{\vec{u}}{\vec{y}} \right]^\alpha \left[\frac{(\mathrm{sub}^\sigma_{v_1}[I'], \{\vec{w_1}\})}{v_1} \ \cdots \ \frac{(\mathrm{sub}^\sigma_{v_n}[I'], \{\vec{w_n}\})}{v_n} \right]$$

(by Lemma 11 and definition of I')

$$\equiv \ \mathrm{Exp}^\Delta[d(I)] \quad \text{(by definition of } \mathrm{Exp}^\Delta\text{)}$$

Figure 4.3: Proof that Exp^Δ is an acceptable expansion.

Chapter 5

On the implementation of notational definition

This chapter discusses a few issues related to the implementation of mechanisms to support notational definitions. It also describes a mechanism that has been implemented based on the formal account developed in the previous chapters. Section 5.1 describes the Nuprl style of declaring new notations. Section 5.2 addresses various approaches to the implementation of a language $\mathcal{L}$ together with a mechanism for notational definition. 'The application of the formal account of notational definition to the reimplementation of Nuprl is briefly discussed in Section 5.3. Section 5.4 describes an implementation of a definition mechanism in the EFS [Gri87].

5.1 Declaration and display of defined notation

This section describes one possible input style for definitional equations, called the Nuprl-style, after the mechanism of that system. A declaration of new notation has the form

$$\begin{array}{lll} \textit{Define name} & & \\ p_1\ s_1\ p_2 \cdots p_n\ s_n\ p_{n+1} & == & rhs \end{array}$$

where *name* is a name associated with the equation, the p_i denote print instructions, and the s_j denote the locations of binding variables or subterms. In Nuprl, the s_i are of the form $\langle u_i \rangle$ where u_i is a variable and the brackets are used to distinguish such variables from the surrounding print instructions. This notation could be extended so that the s_i contain annotations of scoping information. In either case, an algorithm is required to construct, if possible, a σ compatible with $\overline{rhs}$. If such a σ can be constructed, the equation is then interpreted to represent the Δ-equation

$$name(\sigma) \triangleq \overline{rhs}.$$

The s-term σ can then be used as a specification of the binding structure of an instance of this notation

$p_1\ e_1\ p_2 \cdots p_n\ e_n\ p_{n+1}$

where each e_i is a variable or an expression. Such a definition can then be invoked by name, causing the template

$p_1\ \langle\bullet\rangle\ p_2 \cdots p_n\ \langle\bullet\rangle\ p_{n+1}$

to be inserted at an edit point, where the $\langle\bullet\rangle$ are holes that can then filled in by the user using a structure-oriented editor. This is essentially the mode of intantiation used in Nuprl and the EFS (see Section 5.4).

The Nuprl-style of declaration has the advantage of combining the declaration of new notation with the declaration of the display of this notation. This works well for noniterative notations but it is not clear how to extend it in a general way to iterative notations (Section 3.6).

Note that the assumptions regarding the partition of variables used in defining the expansion Exp^{Δ} do not cause any problem in implementing an expansion based on Exp^{Δ}. Variables of the *rhs* of a defining equation could either be of a special lexical class or be internally altered to be so.

5.2 Implementing $\mathcal{L}$

There are primarily two approaches to the implementation of a language $\mathcal{L}$ together with a mechanism for notational definition. The first approach implements the typed λ-calculus with δ- and Δ-equations. Such an implementation could support facilities for the definition of $\mathcal{L}$, with the declaration of a signature for an encoding of the language, and the declaration of new notations. One advantage of this approach is that such an implementation could be used for any language $\mathcal{L}$ that can be encoded using Church's method.

However, it may be desirable to implement only a subset of the expressions $\Lambda[\Sigma^{\mathcal{L}}]$, in particular, $\overline{\mathcal{L}} = \{\overline{A} \mid A \text{ in } \mathcal{L}\}$. The second approach extends an implementation of the language $\mathcal{L}$ with notational definitions. This approach is useful when extending a previously existing implementation of a language $\mathcal{L}$ to one that supports definitions. This is essentially the approach thet was taken in the

Regardless of the approach taken, the formal account of notational definition can be used as a criterion to judge the correctness of a definition mechanism for such an implementation. For example, suppose that rep[e] is a mapping from $\overline{\mathcal{L}}$ to some machine representation of expressions. Then a procedure Ψ operating on this implementation is an acceptable expansion if and only if

$$Exp^{\Psi}[d(I)] \equiv \text{rep}^{-1}[\Psi[\text{rep}[d(I)]]]$$

is an acceptable expansion.

5.3 Applications to Nuprl

As indicated in the Chapter 1, the deficiencies in the definition mechanism of Nuprl originally motivated the work developed in the preceding chapters. The Nuprl system provides a very flexible declaration style that allows users to define new display forms. The problem with this mechanism is that it may be too flexible because it allows for the following definition.

```
DEF rparen4
] == ))))
```

In other words, the text on the right-hand side of a definition is not constrained to be a well-formed term. Each Nuprl term has associated with it a display form that instructs the system on how to display the term to a user. The definition mechanism simply allows a user to define new display forms for terms. In Nuprl terms, instances of defined notations are always fully expanded internally, with the user seeing only the defined display forms. Thus, the Nuprl mechanism does not treat such definitional equations as extending the syntax of the base language in any way and does not recognize any structure (such as binding structure) in an instance of defined notation.

These problems could be overcome by designing a mechanism based on the formal account of notational definition presented above together with a syntactic specification Σ^{Nuprl} for the syntax of the Nuprl language. A fragment of such a specification is given in Figure 5.1. (This syntax will be referred to again in Chapter 7.) The syntactic category of Nuprl expressions is represented by o. There is one additional base type of *integer expressions* N used for universe indices.

The Nuprl system has an interface to the ML programming language that is used for writing programs that manipulate terms and proofs. This interface could be extended to include functions for constructing and analyzing instances of defined notations such as

```
is_def_instance : term -> bool
expand_def_instance : term -> term
name_of_def_instance : term -> tok
```

$$
\begin{array}{rcl}
e & ::= & x \\
 & | & \mathrm{case}(e_1; x.e_2; y.e_3) \\
 & | & \mathrm{inl}(e) \\
 & | & \mathrm{inr}(e) \\
 & | & e_1 + e_2 \\
 & | & \mathrm{spread}(e_1; x, y.e_2) \\
 & | & \langle e, e \rangle \\
 & | & e_1 \wedge e_2 \\
 & | & e_1(e_2) \\
 & | & \lambda x.e \\
 & | & e_1 \rightarrow e_2 \\
 & | & \mathrm{r} \\
 & | & \mathrm{I}(e_1, e_2, e_3) \\
 & | & e_1 = e_2 \in e_3 \\
 & | & \mathrm{U}_i
\end{array}
$$

c_{op}	τ_{op}
case	$o \times (o \rightarrow o) \times (o \rightarrow o) \rightarrow o$
inl	$o \rightarrow o$
inr	$o \rightarrow o$
union	$(o \times o) \rightarrow o$
spread	$o \times (o \rightarrow o \rightarrow o) \rightarrow o$
pair	$(o \times o) \rightarrow o$
$\wedge$	$(o \times o) \rightarrow o$
ap	$(o \times o) \rightarrow o$
lam	$(o \rightarrow o) \rightarrow o$
arrow	$(o \times o) \rightarrow o$
r	o
I	$(o \times o \times o) \rightarrow o$
eq	$(o \times o \times o) \rightarrow o$
U	$\mathrm{N} \rightarrow o$

Figure 5.1: Encoding a fragment of the language of Nuprl.

```
destruct_def_instance : term -> term list
make_def_instance : term list -> term
```

as well as well-behaved functions for substitution, free variable computation, and α-conversion on terms.

5.4 Notational definition in the EFS

The framework developed in the preceding chapters was used by the author in the implementation of a definition mechanism in the EFS[1], the Environment for Formal Systems. This section describes some features of the EFS, including the definition mechanism. A more complete description can be found in [Gri87].

The EFS provides features that allow a user to

- define the syntax and inference rules of a formal system,
- define notational abbreviations,

[1]This section describes EFS version 2, which is available in Release 3.0 of the Synthesizer Generator. EFS version 1, which was in Release 2.0 and documented in [Gri87], was implemented before the analysis of notational definition was complete. In other words, it contains some bugs.

- construct "bottom-up" proofs,
- construct goal-directed "refinement" proofs,
- define new refinement rules that are checked for validity at declaration time, and
- organize theories in a hierarchical structure.

The EFS supports two AUTOMATH [dB80] related formalisms for encoding logical systems: the Edinburgh Logical Framework [HHP87] and the Calculus of Constructions [Coq85,CH85]. These systems employ similar typed λ-calculi with dependent types to encode expressions, rules, and proofs, albeit using quite different approaches. An EFS user can choose to work in the type system of the Edinburgh Logical Framework or in that of the Calculus of Constructions. The common syntax is modeled using the approach outlined in section 2.2, with the same syntactic signature $\Sigma^{\lambda\Pi}$, having only one base type o presented in Figure 5.2.

There is a potential source of confusion that should be addressed. Both the Theory of Constructions and the Edinburgh Logical Framework can be used to encode the syntax of a language $\mathcal{L}$ in a manner similar to that developed in section 2.2. The EFS allows a user to encode such a language interactively and, once defined, to use the definition mechanism to introduce new notations for this language (see for example Figures 5.4 and 5.5 below). It should be noted that these definitions are taking place at the level of the representation $\Sigma^{\lambda\Pi}$, and not some representation $\Sigma^{\mathcal{L}}$ of the language $\mathcal{L}$. The EFS implements only those expressions of type o in normal form, as well as an expansion procedure based on

$$\mathrm{Exp}[d(I)] \equiv \mathrm{nf}^{\beta\eta\pi}[\mathrm{Exp}^{\Delta}[d(I)]]$$

(see Section 3.5). The EFS does not implement a type inference for the level of the encoding $\Lambda[\Sigma^{\lambda\Pi}]$ because no ill-formed expressions can be constructed. The system does, however, support a type inference mechanism for the languages of athe Theory of Constructions and the Edinburgh Logical Framework.

The EFS was implemented with the Cornell Synthesizer Generator [RT84,RT85, RT88b,RT88a]. The Synthesizer Generator is a tool for building full-screen language-based environments from high-level specifications. The Generator was originally designed for implementing language-based environments for programming languages. It is hoped that the EFS will help demonstrate its usefulness in a much wider range of applications. The EFS is among the sample specifications included with the distribution of the Synthesizer Generator, Releases 2.0 and 3.0.

The Synthesizer Generator takes as input a specification written in the Synthesizer Specification Language (SSL) and produces as output a language-based editing environment. An editor specification consists of a series of declarations that define a

$$
\begin{array}{rcl}
e & ::= & x \\
 & | & \text{Type} \\
 & | & \text{Kind} \\
 & | & e(e) \\
 & | & \lambda x.e \\
 & | & \Pi x : e.e
\end{array}
$$

c_{op}	τ_{op}
Type	o
Kind	o
ap	$(o \times o) \rightarrow o$
lam	$(o \rightarrow o) \rightarrow o$
pi	$o \times (o \rightarrow o) \rightarrow o$

Figure 5.2: Encoding for the language of the EFS.

context-free grammar, display information, parsing rules, context-sensitive constraints, and editor-specific commands. An attribute grammar formalism is used to define context-sensitive constraints. The editor produced is a full-screen editor with many features resembling those of Emacs [Sta81]. All generated editors share a common user interface that is extended with editor-specific commands declared in their specification. The reader is referred to the Synthesizer Generator Manual by Reps and Teitelbaum [RT88b] for a complete description of SSL as well as the language-independent commands and how they are invoked.

What distinguishes editors created by the Synthesizer Generator from text editors is that each edited file contains a derivation tree of the context-free grammar defined in the editor's specification. The editor supports multiple buffers, with each buffer containing a single derivation tree. Interacting with the editor amounts to modifying a derivation tree. Each derivation tree is attributed according to the attribute equations of the specification. Attribute values are incrementally updated to achieve a consistent attribution after each modification [DRT81,Rep85]. Such editors have been referred to as a "spread-sheets for trees."

Each editor provides the same set of language-independent commands for manipulating trees, such as cutting and pasting subtrees and moving from one subtree to another. One subtree, called the *current selection*, is highlighted in each buffer and represents the user's focus of interest. The language-specific commands defined in a specification are called *transformations*. The current selection determines which transformations are applicable.

A derivation tree may contain "holes" that represent unexpanded nonterminals of the grammar productions. These "holes" are often displayed as <X> (where X is the unexpanded grammar symbol) and are referred to as *placeholders*. Many transformations are applicable only when the current selection is a placeholder. The act of invoking such a command will be refered to as "inserting a template."

The following conventions regarding type styles will be used. The *italic* style will

Command	*Template*
apply	< term > (< term >)
lambda	\ < *id* >:< *term* > . < *term* >
pi	PI < *id* > : < *term* > . < *term* >
-- >	< term > -- >< term >
type	TYPE
()	(< term >)

Figure 5.3: Templates for basic terms

indicate the name of a command. `Typewriter` style will be used to indicate the EFS display. Since attribute values contribute to the display of a derivation tree, and are not themselves selectable, **boldface** type will be used to distinguish selectable placeholders when a displayed object contains attribute values containing placeholders.

The command *alternate-unparsing-toggle* allows the user to toggle between two different display forms of a derivation tree. The display choice is persistent between a write and a read of a file. Most often this is used to elide the display of a subtree.

Those EFS terms not containing definition instances (see below) will be referred to a *basic terms*. In the following grammar, e ranges over basic terms and x over identifiers:

$$e := x \mid (e) \mid e(e) \mid \backslash x:e.e \mid \mathtt{PI}\ x:e.e \mid e\texttt{-->}e \mid \mathtt{TYPE}.$$

The language described in [HHP87] is defined by a grammar consisting of three levels corresponding to values, types, and kinds. These levels have been collapsed into one and a type inference mechanism implemented that sorts out values, types and kinds. The "-->" constructor has been added for the nondependent function space (A-->B represents PI x:A.B when x is not free in B). The similarity of the languages employed in the LF and CC type systems allows us to use the same grammar for both. It is hoped that the translations to the EFS syntax are obvious.

Placeholders for terms and identifiers are displayed as <term> and <id> respectively. Commands for inserting templates at a <term> placeholder are shown in Figure 5.3. Basic terms and identifiers can also be typed directly and parsed when the current selection is a <term> or <id> placeholder. A "?" parses to <term>. Entering "? --> ?" will result in the expression <term> --> <term> being inserted.

Each EFS file consists of a list of named *chapters*. Each chapter contains a list of *declarations*. Each declaration occurs in the context of a set of *environments*. There are four such environments :

- The *constant environment* associates constants with their types and is extended by a constant declaration.
- The *value environment* associates a name with a value of some type. It is extended by a value declaration or a refinement.
- The *definition environment* associates a name with a syntactic template.
- The *rule environment* associates a name with a user-defined refinement rule.

Refinement and rule declarations will not be discussed here. A constant declaration associates an identifier with a type. The *constant* command transforms a <declaration> placeholder to a template and is displayed as:

```
CONSTANT <id> : <term>
```

A constant declaration is well-formed if both the identifier and term are complete, the identifier is unique in the current environment, and the term is a well-formed type expression. A binding is not added to the constant environment unless the constant declaration is well-formed.

The declaration of a few syntactic constants and the basic judgment form are shown in Figure 5.4. (The definition of first-order terms and rules used in this section follows closely the one given in [HHP87].)

Definition declarations allow for the interactive creation of new templates that function as notational abbreviation for terms. The declaration style has been taken from the definition mechanism that has proved to be indispensable in the Nuprl system (although the implementations differ). The <declaration> placeholder can be transformed to a definition declaration by the *definition* command:

```
DEFINITION <id>
<lhs> == <term>
```

The left-hand side defines the new template as abbreviating the term on the right-hand-side. A completed definition declaration has the form:

DEFINITION defname
s_1 $\langle v_1 \rangle$ s_2 $\langle v_2 \rangle$ ... s_n $\langle v_n \rangle$ s_{n+1} == T

The s_i are strings (possibly null) and the $\langle v_i \rangle$ designate parameters. (This is the only place where the angle brackets do not signify a placeholder, but they are used here to distinguish parameters from surrounding text.) Figure 5.5 contains defined templates for the constants of Figure 5.4.

```
CONSTANT o : TYPE

CONSTANT i : TYPE

CONSTANT Absurd : o

CONSTANT Equality : i --> (i --> o)

CONSTANT And : o --> (o --> o)

CONSTANT Implies : o --> (o --> o)

CONSTANT ForAll : (i --> o) --> o

CONSTANT Exists : (i --> o) --> o

CONSTANT True : o --> TYPE
```

Figure 5.4: Constant declarations.

The left-hand side of a definition declaration can be entered in its entirety and parsed. If an s_i is to contain any character from the set {~,<,>} it must escaped with the ~ character on entry. The tokens {~n,~t,~b} have special meaning on the left-hand side of a definition declaration. The ~t moves to the current left margin on the next line, while ~t and ~b move the current left margin left and right one indentation stop respectively. For example, the left-hand side of the definition declaration

```
DEFINITION let
let <x> = <a> :  <b>
in
 <c>
ni  == (\x:b.c)(a)
```

was produced by typing

```
let <x> = <a>:<b>~nin~t~n<c>~b~nni
```

when the <lhs> placeholder was selected.

Invoking a definition is accomplished in one of two ways:

1. Typing "@defname" when positioned at a <term> placeholder. This will result in an instance of this definition with all of the <v_i> replaced by <term> or <id> placeholders.

```
DEFINITION eq
<a> = <b> == Equality(a)(b)

DEFINITION and
<a> & <b> == And(a)(b)

DEFINITION implies
<a> => <b> == Implies(a)(b)

DEFINITION not
~<a> == (a => Absurd)

DEFINITION all
all <x>.<a> == All(\x:i.a)

DEFINITION some
some <x>.<a> == Exists(\x:i.a)

DEFINITION true
<a> true == True(a)
```

Figure 5.5: Definitions for basic constants.

2. Typing "`@(defname` $\texttt{t}_1$`,`$\texttt{t}_2$`, ... ,`$\texttt{t}_k$`)`" when positioned at a <term> placeholder. Here the $\texttt{t}_i$ represent terms or definition invocations. This will result in an instance of the definition with the first k slots filled in with the $\texttt{t}_i$ from left to right.

For example, typing "`@and`" in the context of the definitions of Figure 5.5 will cause the following template

    <term> & <term>

to be inserted, whereas typing "`@(and A (@(implies B)))`" will result in the term

    A & (B => <term>)

being inserted. Currently, there are no facilities for updating the parser to recognize these defined terms, leaving this as the only method of entering them.

It may be necessary to ascertain which definition a term is an instance of. Selecting a definition instance `d` and invoking the *alternate-unparsing-toggle* command will change its display to `[name:d]` where `name` is the definition's name. This is particularly useful for differentiating between instances of two definitions having the same display.

Invoking a definition that contains a binding variable will result in a template with all binding variable slots replaced by <id>. For example, invoking the `all` definition of Figure 5.5 will produce the template `all <id>.<term>`. The procedures for computing the free variables of a term, substitution, and α-conversion can work directly on a definition instance without expanding the definition. For example, the term

    \w:i.(\x:i.all w.x = w)(w)

would β-reduce to the term

    \w:i.all w'.w = w'

where the instance of the definition `all` has been α-converted to avoid capture.

Chapter 6

Tactic Trees

The standard account of Nuprl [BC85,CKB85,Con86] describes the deductive apparatus of refinement in three stages. First, a *refinement logic* is presented that has its rules arranged to directly support goal-directed reasoning. These rules are called *refinement rules* and are presented in a top-down style. A refinement proof in Nuprl is a tree. Leafs contain unproven goals, while each interior node contains a goal and a rule and has as children nodes containing goals that correspond to subgoals of the rule. The refinement editor of Nuprl allows a user to interactively construct a refinement proof in a step-by-step fashion by selecting unproven subgoals and invoking inference rules. When a rule is applied to a goal the system generates the corresponding subgoals.

Second, a separate mechanism that *extracts* terms from refinement trees is described. If type-theoretic assertions are viewed as a program specifications, then these extracted terms can be viewed as a functional programs meeting these specifications.

Third, this apparatus is extended with the LCF tactics [GMW79] by encoding this refinement logic in the so-called metalanguage ML. This was explained in the following statement about λ-Prl (the immediate predecessor to Nuprl), quoted from [CKB85].

> During the design of λ-PRL we chose to integrate it with ML because we believed that the Edinburgh LCF approach to automating reasoning was especially compatible with our design principles. For instance, the PRL logic is oriented to top-down development, as are LCF tactics. ... Our previous experience with LCF convinced us that the tactic concept was not only viable and flexible, but that it would be feasible to incorporate into our implementation.

This is done in such a way that rules may be either primitive refinement rules, or user-defined rules in the form of ML tactics. Thus, the tactics framework is presented as an extension to the basic apparatus of refinement. It enables users to avoid much of the tedium associated with formal proofs through the definition of derived rules, proof search strategies, and decision procedures.

Of course, this is a perfectly adequate and comprehensible account of refinement. However, several questions naturally arise. Can the refinement style of proof construction support proof development in any logic, or is this method of proof construction limited in applicability only to "refinement logics" such as the logic of Nuprl? What exactly is a refinement logic? What logic does the Nuprl refinement system implement?

This chapter presents an account called *abstract refinement* that not only addresses these questions but provides a uniform and logic-independent treatment of the three aspects of refinement described above. The main ideas can be briefly stated here. The account begins with the LCF notion of encoding formal systems with a collection of tactics. A logic-independent account of refinement is developed by extending the LCF tactics framework with a new construct called a *tactic tree.* Tactic trees are tree-structured objects representing the iterated composition of tactics. With the definition of tactic trees the notions of refinement and of a refinement logic can be viewed as derived ones in the sense that tactic trees constructed over a collection of tactics naturally give rise to a *refinement system.* A desired *refinement logic* is obtained by providing the appropriate collection of tactics.

The following chapter presents a detailed discussion concerning the construction of proofs with tactic trees and the manner in which the refinement logic as well as extraction of Nuprl refinement can be modeled within the framework of tactic trees.

The tactic-tree approach has several advantages. First, it is a simpler account of refinement. It arises form a natural extension to the LCF tactics framework and, as mentioned, gives a uniform treatment to the three separate aspects of refinement described above. Second, the tactic-tree approach is logic-independent and makes the Nuprl refinement style of proof development applicable to a wide range of problem areas and logics. A tactic tree component can be implemented independently of a particular application. Third, it simplifies the task of explaining the relationship of a logic to its various refinement presentations, and in this way provides a clear conception of refinement by giving it a more direct proof-theoretic justification. This can increase confidence that an implementation of a logic is correct, and provide a precise definition of the relationship of a refinement logic to conventional proof-theoretic concepts.

This chapter is organized as follows. Section 6.1 defines the framework of tactics and tacticals based on *goals*, *events*, and an *achievement relation* taken from LCF [GMW79]. The development of tactics and tactic trees will be presented in the abstract setting of set theory, rather than in the programming language ML as is usually done. In Section 6.2 the definition of *tactic trees* is introduced. Section 6.3 addresses some issues related to the implementation of tactics and tactic trees.

6.1 Tactics.

This section presents the LCF tactics framework of [GMW79]. It is presented in set theory rather than in some dialect of the programming language ML as is usually done. This approach is taken for several reasons. It provides an *idealized* setting where the essential ideas can be presented without the complications that a programming language presentation would require. Second, as will be discussed in Chapter 7, this presentation makes it easier to treat proof-theoretic considerations such as the soundness and completeness of a collection of tactics with respect to a formal system.

The following set-theoretic notations will be used. If A and B are sets, then their disjoint union $\{\mathrm{inl}(a) \mid a \in A\} \cup \{\mathrm{inr}(b) \mid b \in B\}$ is denoted by $A + B$. The set of n-ary tuples over A $\{(a_1, \ldots, a_n) \mid a_i \in A\}$ is denoted by A^n. The 0-ary tuple is written as $()$. The set of functions from A to B is denoted by $A \rightarrow B$. Functions will be written using the λ-notation, $\lambda x.b$. If f is a function and then $f(a)$ denotes the value of that function applied to a. The application of a function to a tuple, $f((a_1, \ldots, a_n))$, is written as $f(a_1, \ldots, a_n)$, and so $f()$ stands for $f(())$.

A *problem space* is a triple $(\mathcal{G}, \mathcal{E}, \mathcal{A})$, where $\mathcal{G}$ and $\mathcal{E}$ are sets and $\mathcal{A}$ is a binary relation, $\mathcal{A} \subseteq \mathcal{E} \times \mathcal{G}$, is called The set $\mathcal{G}$ is called the set of *goals*, $\mathcal{E}$ the set of *events*, and $\mathcal{A}$ the *achievement relation.* The intuition behind this formalism is that the members of $\mathcal{G}$ are thought of as representing *problems* and the members of $\mathcal{E}$ as representing a *potential solutions.* A goal g is solved by an event e if and only if $\mathcal{A}(e, g)$. Such an e is said to *achieve* the goal g.

A *refinement* of a goal $g \in \mathcal{G}$ is a tuple of the form

$$((g_1, \ldots, g_n), v),$$

where each $g_i \in \mathcal{G}$, and v is a function $v \in \mathcal{E}^n \rightarrow \mathcal{E}$ that is called a *validation.* Such a refinement of g is called a *valid refinement* if for every n-tuple

$$(e_1, \ldots, e_n) \in \mathcal{E}^n,$$

$\mathcal{A}(e_i, g_i)$ for $1 \leq i \leq n$ implies that $\mathcal{A}(v(e_1, \ldots, e_n), g)$. A valid refinement of g represents the decomposition of the problem g into n derived problems $g_1, \ldots, g_n$. The validation v represents a means of constructing a solution to the original problem g from solutions to the derived problems.

Given a problem space $(\mathcal{G}, \mathcal{E}, \mathcal{A})$, a *tactic* is a refinement-valued function on $\mathcal{G}$. The set of tactics over this problem space is denoted by

$$\mathcal{T}^{(\mathcal{G},\mathcal{E},\mathcal{A})} = \mathcal{G} \rightarrow \bigcup_{0 \leq n} (\mathcal{G}^n \times (\mathcal{E}^n \rightarrow \mathcal{E})).$$

A tactic $T \in \mathcal{T}^{(\mathcal{G},\mathcal{E},\mathcal{A})}$ is *valid* (with respect to the problem space $(\mathcal{G}, \mathcal{E}, \mathcal{A})$), if for all $g \in \mathcal{G}$,

$$T(g) = ((g_1, \ldots, g_n), v)$$

is a valid refinement of g. Define the set

$$\mathcal{VT}^{(\mathcal{G},\mathcal{E},\mathcal{A})} \subset \mathcal{T}^{(\mathcal{G},\mathcal{E},\mathcal{A})}$$

to be the set of valid tactics (w.r.t. the problem space $(\mathcal{G},\mathcal{E},\mathcal{A})$). This set is called simply the set of valid tactics when $\mathcal{A}$, $\mathcal{G}$, and $\mathcal{E}$ are clear from the context.

If T is a valid tactic and $T(g) = ((), v)$, then T is said to *achieve* the goal g, since $\mathcal{A}(v(), g)$.

A very simple example of a valid tactic is

$$\text{idtac} = \lambda g.((g), \lambda x.x)$$

which just returns its argument together with the identity validation.

A function

$$f \in (\mathcal{T}^{(\mathcal{G},\mathcal{E},\mathcal{A})})^n \rightarrow \mathcal{T}^{(\mathcal{G},\mathcal{E},\mathcal{A})}$$

is called an n-ary tactical, or simply a tactical. Such a tactical is *valid* if for every collection of n valid tactics $T_1, \ldots, T_n$ then $f(T_1, \ldots, T_n)$ is a valid tactic.

An important family of valid tacticals is made up of the composition tacticals

$$\text{compose}^n \in (\mathcal{T}^{\mathcal{G},\mathcal{E}})^{n+1} \rightarrow \mathcal{T}^{\mathcal{G},\mathcal{E}}$$

defined as

$$\begin{aligned}
&\text{compose}^n(T_0, T_1, \ldots, T_n)(g) = \\
&\quad ((g^1_1, \ldots, g^1_{m_1}, \ldots, g^n_1, \ldots, g^n_{m_n}), v) \\
&\quad \text{where } v = \lambda \vec{x_1}, \ldots, \vec{x_n}.v_0((v_1(\vec{x_1}), \ldots, v_n(\vec{x_n}))), \\
&\quad T_0(g) = ((g^0_1, \ldots, g^0_n), v_0), \\
&\quad T_i(g^0_i) = ((g^i_1, \ldots, g^i_{m_i}), v_i) \ \text{ for } 1 \leq i \leq n, \\
&\quad \text{and } \vec{x_i} = x^i_1, \ldots, x^i_{m_i}, \\
&\quad \text{where } m_i \text{ is the arity of } v_i.
\end{aligned}$$

It is easy to check that compose^n is a valid $(n+1)$-ary tactical. In what follows, this collection of tacticals will be denoted as

$$\mathcal{C} = \bigcup_{n \leq 0} \{\text{compose}^n\}.$$

Note that compose$^0(T)$ is T.

Definition 28 ($\mathcal{TS}^{(\mathcal{G},\mathcal{E},\mathcal{A})}(\mathcal{B},\mathcal{TC})$) *Given a problem space $(\mathcal{G},\mathcal{E},\mathcal{A})$, a set of tactics $\mathcal{B} \subset \mathcal{T}^{(\mathcal{G},\mathcal{E},\mathcal{A})}$, and a set of tacticals*

$$\mathcal{TC} \subseteq \bigcup_{1 \leq n} \left((\mathcal{T}^{(\mathcal{G},\mathcal{E},\mathcal{A})})^n \rightarrow \mathcal{T}^{(\mathcal{G},\mathcal{E},\mathcal{A})} \right),$$

inductively define the set of tactics $\mathcal{TS}^{(\mathcal{G},\mathcal{E},\mathcal{A})}(\mathcal{B},\mathcal{TC})$, called a tactic system, *or the* tactics generated by $\mathcal{B}$ and $\mathcal{TC}$, *as the closure of $\mathcal{B}$ under $\mathcal{TC}$. That is,*

1. *if* $T \in \mathcal{B}$ *then* $T \in \mathcal{TS}^{(\mathcal{G},\mathcal{E},\mathcal{A})}(\mathcal{B},\mathcal{TC})$,
2. *if* $T_1,\ldots,T_k \in \mathcal{TS}^{(\mathcal{G},\mathcal{E},\mathcal{A})}(\mathcal{B},\mathcal{TC})$ *and* $f \in \mathcal{TC}$ *is a k-ary tactical, then* $f(T_1,\ldots,T_k) \in \mathcal{TS}^{(\mathcal{G},\mathcal{E},\mathcal{A})}(\mathcal{B},\mathcal{TC})$.

Fact 13 *If* $\mathcal{B}$ *is a set of valid tactics (w.r.t.* $(\mathcal{G},\mathcal{E},\mathcal{A})$*) and* $\mathcal{TC}$ *is a set of valid tacticals (w.r.t.* $(\mathcal{G},\mathcal{E},\mathcal{A})$*), then*

$$\mathcal{TS}^{(\mathcal{G},\mathcal{E},\mathcal{A})}(\mathcal{B},\mathcal{TC}) \subseteq \mathcal{VT}^{(\mathcal{G},\mathcal{E},\mathcal{A})}.$$

In other words, the tactics generated by $\mathcal{B}$ and $\mathcal{TC}$ are all valid.

6.2 Tactic Trees

The set of composition tacticals $\mathcal{C}$ provides a functional means of composing tactics. A new construct, called a *tactic tree*, is now introduced that, in some sense, provides a tree-structured representation for the iterated composition of tactics.

Definition 29 (tactic trees over $\mathcal{B}$ with root g)
Given a problem space $(\mathcal{G},\mathcal{E},\mathcal{A})$ *and a set of tactics*

$$\mathcal{B} \subseteq \mathcal{T}^{(\mathcal{G},\mathcal{E},\mathcal{A})},$$

inductively define the set $\mathcal{TT}_g^{\mathcal{B}}$ *of* tactic trees over $\mathcal{B}$ with root g *as*

1. *if* $g \in \mathcal{G}$ *then*
$$\text{leaf}(g) \in \mathcal{TT}_g^{\mathcal{B}},$$
2. *if* $g \in \mathcal{G}$*, for each* $i \in \{1,\ldots,n\}$ $r_i \in \mathcal{TT}_{g_i}^{\mathcal{B}}$, $T \in \mathcal{B}$*, and*
$$T(g) \;=\; ((g_1,\ldots,g_n,v)),$$
then
$$\text{node}(g,T,(r_1,\ldots,r_n),v) \in \mathcal{TT}_g^{\mathcal{B}}.$$

The metavariables $r, r_0, r_1, \ldots$ will be used to denote tactic trees.

Definition 30 ($\mathcal{TT}^{\mathcal{B}}$) *Given a problem space* $(\mathcal{G},\mathcal{E},\mathcal{A})$ *and a set of tactics*

$$\mathcal{B} \subseteq \mathcal{T}^{(\mathcal{G},\mathcal{E},\mathcal{A})},$$

define the set $\mathcal{TT}^{\mathcal{B}}$*, called the set of tactic trees over the set of tactics* $\mathcal{B}$*, to be*

$$\mathcal{TT}^{\mathcal{B}} = \bigcup_{g\in\mathcal{G}} \mathcal{TT}_g^{\mathcal{B}}.$$

If $r \in \mathcal{TT}^{\mathcal{B}}$, then r is called a *tactic tree over* $\mathcal{B}$, or simply a *tactic tree* when $\mathcal{B}$ can be inferred from context.

It should be noted that the presence of validations v in tactic tree nodes is not actually essential to this definition since they can always be recovered by applying the tactic at a node to the root goal at that node. These validations are a part of the definition to make certain aspects of the development clearer.

Note that each leaf of a tactic tree r is of the form

$$\text{leaf}(g)$$

or of the form

$$\text{node}(g, T, (), v).$$

The former will be called an *incomplete leaf* and the latter a *complete leaf* of r. A tactic tree r is *complete* if every leaf of r is a complete leaf.

A tactic tree $r \in \mathcal{TT}^{\mathcal{B}}$ is *valid*, if every tactic T used to construct r is valid. Of course, if $\mathcal{B} \subseteq \mathcal{VT}^{(\mathcal{G},\mathcal{E},\mathcal{A})}$, then every $r \in \mathcal{TT}^{\mathcal{B}}$ is a valid tactic tree.

For a given tactic tree r, define the tactic tsyn(r) that, loosely speaking, transforms the nodes of r into compositions.

Definition 31 (tsyn(r)) *Given a tactic tree* $r \in \mathcal{TT}^{\mathcal{B}}$, *define the function* tsyn(r) *by induction on* r *as*
Case 1: $r = \text{leaf}(g)$, *then* $\text{tsyn}(r) = \text{idtac}$
Case 2: $r = \text{node}(g, T, (r_1, \ldots, r_n), v)$ *for some* $n \geq 0$, *then*

$$\text{tsyn}(r) = \text{compose}^n(T, \text{tsyn}(r_1), \ldots, \text{tsyn}(r_n)).$$

Define the function fringe(r) that takes tactic trees to n-tuples over $\mathcal{G}$.

Definition 32 (fringe(r)) *by induction on* r *as*
Case 1: $r = \text{leaf}(g)$, *then* $\text{fringe}(r) = (g)$
Case 2: $r = \text{node}(g, T, (r_1, \ldots, r_n), v)$ *for some* $n \geq 0$, *then*

$$\text{fringe}(r) = (s_1^1, \ldots, s_{m_1}^1, \ldots, s_1^n, \ldots, s_{m_n}^n)$$

where $\text{fringe}(r_i) = (s_1^i, \ldots, s_{m_i}^i)$.

The elements of the tuple fringe(r) are the roots of incomplete leafs of the tactic tree r in left to right order. Note that if r is a complete tactic tree then $\text{fringe}(r) = ()$.

Define the function root(r) taking tactic trees to goals as

$$\begin{aligned} \text{root}(\text{leaf}(g)) &= g \\ \text{root}(\text{node}(g, T, (r_1, \ldots, r_n), v)) &= g. \end{aligned}$$

The following facts are direct corollaries of these definitions.

Fact 14 *If $r \in \mathcal{TT}^{\mathcal{B}}$, then there exists a validation v such that*

$$\text{tsyn}(r)(\text{root}(r)) = (\text{fringe}(r), v).$$

Fact 15 *If $r \in \mathcal{TT}^{\mathcal{B}}$ is a valid tactic tree, then* $\text{tsyn}(r)$ *is a valid tactic.*

Thus, a tactic tree r with $\text{root}(r) = g$ can be seen as representing a refinement of the goal g.

That tactic trees are a tree-structured representation for the iterated composition of tactics can be made precise by the following characterization.

Fact 16 *If $\mathcal{B} \subseteq \mathcal{T}^{(\mathcal{G},\mathcal{E},\mathcal{A})}$ and $r \in \mathcal{TT}^{\mathcal{B}}$, then*

$$\text{tsyn}(r) \in \mathcal{TS}^{(\mathcal{G},\mathcal{E},\mathcal{A})}(\mathcal{B} \cup \{\text{idtac}\}, \mathcal{C}).$$

In addition, if r is complete, then

$$\text{tsyn}(r) \in \mathcal{TS}^{(\mathcal{G},\mathcal{E},\mathcal{A})}(\mathcal{B}, \mathcal{C}).$$

Note that if r is a complete and valid tactic tree, then

$$\text{tsyn}(r)(\text{root}(r)) = ((), v)$$

and

$$\mathcal{A}(v(), \text{root}(r)).$$

In this case define $\text{syn}(r)$ to be the event $v()$. Alternatively, this function can be defined directly as the composition of the validations of r in a "bottom-up" fashion.

Definition 33 ($\text{syn}(r)$) *Define the function* $\text{syn}(r)$ *taking complete tactic trees to events in $\mathcal{E}$ as*

$$\begin{array}{lcl} \text{syn}(\text{node}(g, T, (), v)) & = & v() \\ \text{syn}(\text{node}(g, T, (r_1, \ldots, r_n), v)) & = & v(\text{syn}(r_1), \ldots, \text{syn}(r_n)) \end{array}$$

A complete tactic tree $r \in \mathcal{TT}^{\mathcal{B}}$ is said to *achieve* a goal $g \in \mathcal{G}$, if and only if $\text{root}(r) = g$ and $\mathcal{A}(\text{syn}(r), g)$.

The function $\text{syn}(r)$ will play an important role in the modeling of Nuprl extraction that is presented in the next chapter.

Tactic trees also provide an abstract and logic-independent model of the kind of development that takes place in the Nuprl refinement editor[1]. That is, tactic trees can be constructed in a step-by-step fashion where each step applies a tactic to the goal of an incomplete leaf and extends the tree with an additional refinement.

[1]Although it will be seen in the next chapter that the tactic-tree account is not so much a generalization of Nuprl refinement as it is an alternative account of it.

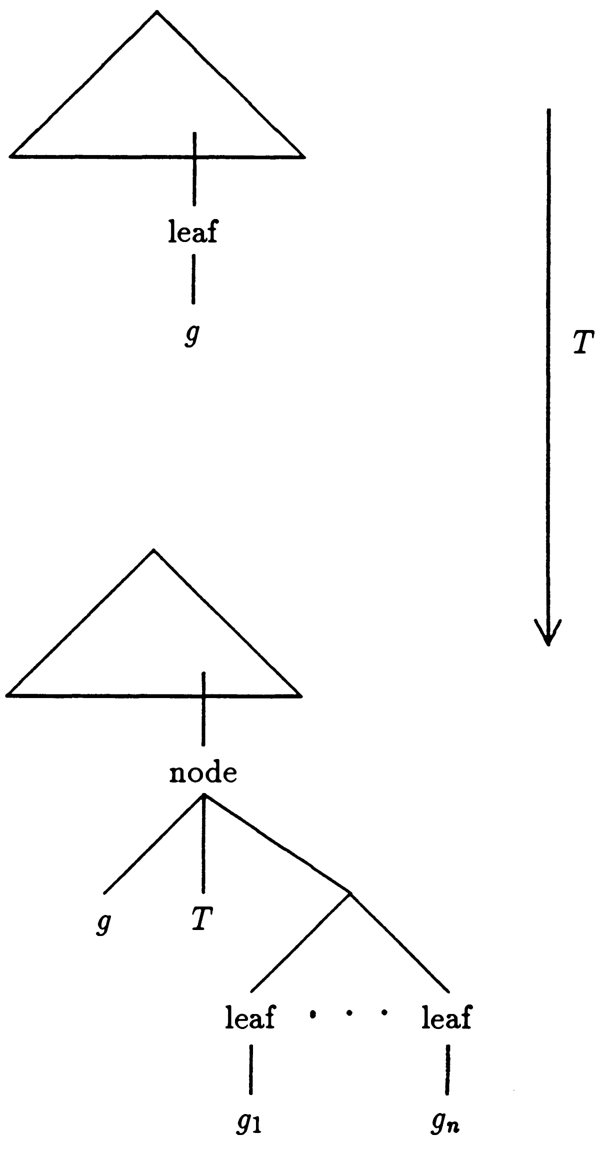

Figure 6.1: A tactic step using $T(g) = ((g_1, \ldots, g_n), v)$.

Definition 34 (A tactic step from r to r'.) *The tactic tree r' is said to be obtained from the tactic tree r* by a tactic step *if r' is the result of replacing some leaf of r of the form* leaf(g) *with*

$$\text{node}(g, T, (\text{leaf}(g_1), \ldots, \text{leaf}(g_n)), v),$$

where T is a tactic and

$$T(g) = ((g_1, \ldots, g_n), v).$$

Note that n may be 0, in which case the complete leaf node$(g, T, (), v)$ *replaces* leaf(g).

This is called a *valid tactic step* from r to r' if T is a valid tactic. A tactic step is presented in Figure 6.2 (ignoring validations).

This can be generalized by defining *transformations* on tactic trees. Given a set of tactics $\mathcal{B}$, define the set of *tactic tree transformations* as

$$\mathcal{TR}^{\mathcal{B}} = \mathcal{TT}^{\mathcal{B}} \rightarrow \mathcal{TT}^{\mathcal{B}}.$$

A function $f \in \mathcal{TR}^{\mathcal{B}}$ is called a *valid transformation* if for any valid tactic tree $r \in \mathcal{TT}^{\mathcal{B}}$, $f(r)$ is also a valid tactic tree with $\text{root}(r) = \text{root}(f(r))$. For example, the following are two very simple valid transformations:

$$\begin{aligned} \text{idtrans} &= \lambda r.r, \\ \text{chop} &= \lambda r.\text{leaf}(\text{root}(r)). \end{aligned}$$

Definition 35 (A step from r_1 to r_2)
The tactic tree r_2 is said to be obtained from the tactic tree r_1 by a step *if r_2 is the result of replacing a proper subtree r_3 of r_1 with $f(r_3)$ for some transformation such that* $\text{root}(r_3) = \text{root}(f(r_3))$. *This is called a valid step if f is a valid transformation.*

Note that if r_1 is a valid tactic tree and r_2 is obtained from r_1 by a valid step, then r_2 is a valid tactic tree.

Given a tactic T, define the transformation mknode(T) as follows:

$$\begin{aligned} &\text{mknode}(T)(r) = \text{node}(g, T, (\text{leaf}(g_1), \ldots, \text{leaf}(g_n)), v) \\ &\qquad \text{where } T(g) = ((g_1, \ldots, g_n), v), \\ &\qquad \text{and } g = \text{root}(r). \end{aligned}$$

Any tactic step from r to r' using the tactic T can then be accomplished by a step using mknode(T).

A sequence of valid tactic trees, $r_0, r_1, \ldots, r_k$, is called a *valid development*, if r_0 is a valid tactic tree and each r_{i+1} is obtained from r_i by a valid step. Note that if $r_0, r_1, \ldots, r_k$ is a valid development then r_k is a valid tactic tree. Such a sequence is called a *complete development* if r_k is complete. Also note that if $r_0, r_1, \ldots, r_k$ is a complete and valid development with $r_0 = \text{leaf}(g)$, then $\mathcal{A}(\text{syn}(r_k), g)$.

6.3 Some implementation issues

This section will discuss some issues related to the implementation of tactics and tactic trees and present a rough sketch of a proposed logic-independent implementation. This section is necessarily tentative because nothing has been implemented at the time of writing. Needless to say, a great deal of work remains to be done in this area.

A logic-independent component supporting tactic trees is imagined to be made up of three parts: an implementation of tactics and tactic trees, a *logic specification interface*, and a *user interface.*

The logic specification interface is to be used by a *logic implementor* for defining a problem space and a basic collection of tactics and tacticals. The logic implementor would also be responsible for declaring display and parsing information for various constructs.

If a formal system is being implemented, the logic implementor will presumably have undertaken an analysis, similar to those presented in the next chapter, resulting a tactic system $\mathcal{TS}^{(\mathcal{G},\mathcal{E},\mathcal{A})}(\mathcal{B},\mathcal{C})$, which can then guide the actual implementation of these tactics (representation in a programming language). The logic implementor may extend this collection by defining new tacticals. It is the implementor's responsibility to ensure that any such tacticals are valid and conservatively extend (see the next chapter for a definition) the basic system of tactics.

The user interface is imagined to be much like the one provide by RED (the Refinement EDitor) of Nuprl. It would support a window-based graphics interface to tactic trees and provides the *user* with commands for viewing, navigating, and modifying tactic trees. Modification would include, of course, the application of tactics and transformation functions.

The entire tactic-tree component is envisioned as a part of a larger system that includes a library manager. A library would contain, among other things, objects containing the declaration of defined notation, objects containing the declaration of tactics and transformations, and objects containing tactic trees. However, these other components will not be discussed here.

For the purposes of this discussion, assume that the system is implemented in some dialect of ML. The programming language ML was originally developed as the so-called metalanguage for the Edinburgh LCF [GMW79] where the tactics framework was first presented. The original LCF system used ML to represent tactics and to construct derivations in a formal system called $PP\lambda$. The LCF approach has been applied to many other logics and plays a central role in Nuprl and other systems such as Isabelle [Pau86], the Göteborg group's implementation of type theory [Pet82,NPS86]. Several features of the language were designed to make this easy and natural: it is strongly typed, it is a higher-order functional language, and has an exception mechanism. Subsequently, ML has evolved into an important general purpose programming language in its own right [HMM86]. The reader is assumed to be somewhat familiar with some dialect of ML, and so no overview of the language is presented here. Typewriter font will be used to indicate ML language constructs.

The logic implementor would define the types

```
goal = ...
event = ...
```

from which the type `tactic` might then be defined by the system as

```
tactic = goal -> (goal list # validation)
```

where

```
validation = event list -> event.
```

This encoding uses the ML exception mechanism to handle tactic failure, and uses the `list` type to encode n-ary tuples.

One problem with this definition is that it is fairly easy to write functions of type `tactic` that are not valid. This problem can not be avoided, as was remarked in [GMW79] (p. 57),

> ... it is hard to see how to design a programming language so that all definable objects of type tactic are valid, or how to gain this effect by a type discipline. At most we can adopt a style which encourages the programming of valid tactics.

One way of encouraging this style is by the use of valid tacticals. In addition, the system could use an alternative definition of the type `tactic`, such as

```
tactic = goal -> tactic-result,
```

where `tactic-result` is an abstract data type with one constructor

```
make-tactic-result :
          (goal list # validation) -> tactic-result
```

and two destructors

```
validation-of-tactic-result : tactic-result -> validation
subgoals-of-tactic-result : tactic-result -> goal list
```

The system could then allow the logic implementor access to all three functions, while allowing the user access only to the last function. If the logic implementor used these "private" functions only in the definition of valid tactics and tacticals, then this would seem to ensure that the result of applying any *user-defined* tactic would be a valid refinement for some goal, although not necessarily the goal to which the tactic was applied.

This motivates the following definitions (returning to the set-theoretic notation):

Definition 36 (**$\mathcal{B}$ is almost valid**) *Given a problem space $(\mathcal{G}, \mathcal{E}, \mathcal{A})$, a set of tactics $\mathcal{B} \subset \mathcal{T}^{(\mathcal{G},\mathcal{E},\mathcal{A})}$ is* almost valid *if for all $T \in \mathcal{B}$ and for all $g \in \mathcal{G}$, if $T(g) = ((g_1, \ldots, g_n), v)$, then $((g_1, \ldots, g_n), v)$ is a valid refinement for some $g' \in \mathcal{G}$. That is, for every n-tuple $(e_1, \ldots, e_n) \in \mathcal{E}$ with $\mathcal{A}(e_i, g_i)$ for $1 \leq i \leq$, $\mathcal{A}(v(e_1, \ldots, e_n), g')$.*

Definition 37 (**a consistency relation for a problem space**)
A consistency relation *for a problem space $(\mathcal{G}, \mathcal{E}, \mathcal{A})$ a relation $\mathcal{K} \subseteq \mathcal{E} \times \mathcal{G}$ such that if $((g_1, \ldots, g_n), v)$ is a valid refinement for some goal in $\mathcal{G}$ and for all $(e_1, \ldots, e_n) \in \mathcal{E}^n$ with $\mathcal{A}(e_i, g_i)$ for $1 \leq i \leq n$, $\mathcal{K}(v(e_1, \ldots, e_n), g)$ implies $\mathcal{A}(v(e_1, \ldots, e_n), g)$.*

Fact 17 *If $\mathcal{B} \subset \mathcal{T}^{(\mathcal{G},\mathcal{E},\mathcal{A})}$ is an almost valid set of tactics and $\mathcal{K}$ is a consistency relation for $(\mathcal{G}, \mathcal{E}, \mathcal{A})$, then, for any $T \in \mathcal{B}$ and $g \in \mathcal{G}$ with $T(g) = ((), v)$, if $\mathcal{K}(v(), g)$ then $\mathcal{A}(v(), g)$.*

The point is that $\mathcal{A}$ may be undecidable while $\mathcal{K}$ is decidable. If $\mathcal{K}$ is decidable the implementor could define the function

```
is_consistent : event # goal -> bool = ...
```

encoding this relation. This function could be used with an almost valid collection of tactics to check that achievement holds. In particular, this function could then be used to check that a tactic tree is *consistent* is the following sense (again, returning to the set-theoretic notation):

Definition 38 (**r is consistent**) *A tactic tree r is* consistent *if for each complete subtree r' of r, $\mathcal{A}(\text{syn}(r'), \text{root}(r'))$.*

So, if r is a complete and consistent tactic tree, then $\mathcal{A}(\text{syn}(r), \text{root}(r))$. Note that this holds for complete and valid tactic trees since if r is a valid, then r is consistent.

The implementation of tactic trees could then incrementally check consistency of a tactic tree, constructed over a set of almost valid tactics, as subtrees are completed. This could be done in a bottom-up fashion using the function `is_consistent` together with an implementation of the function syn. One possible implementation would use attribute values at tactic-tree nodes and incrementally propagate this information as the tree is modified [DRT81,Rep85].

Of course, such an implementation would include a large collection of predefined valid tacticals and valid transformation functions. Among these would be the following standard tacticals (taken from Nuprl [Con86] and LCF [GMW79]):

`THENL` If `L` is a list of tactics, then the tactic `(T THENL L)` first applies the tactic `T` and then applies the tactics in `L` to the resulting subgoals in a left to right order. This essentially implements the entire family of set-theoretic tacticals $\mathcal{C}$.

`THEN` The tactic `(T THEN T')` first applies the tactic `T` and then applies `T'` to each resulting subgoal.

`ORELSE` This tactical provides a means of backtracking.
The tactic `(T ORELSE T')` first attempts to apply the tactic `T` to a goal. If this is successful the result is returned. If it fails the result is that of applying `T'`.

`REPEAT` The tactic `(REPEAT T)` first applies the tactic `T` and then repeatedly applies `(REPEAT T)` to all subgoals obtained until there is a failure. The tactic `(REPEAT T)` itself never fails.

`COMPLETE` The tactic `(COMPLETE T)` returns the result of applying the tactic `T` only if this contains the empty list of subgoals, otherwise it fails.

`PROGRESS` The tactic `(PROGRESS T)` returns the result of applying the tactic `T` only if this actually changes the goal, otherwise it fails.

The implementation also could include functions for transformations that correspond to `THENL`, `THEN`, and `ORELSE`. (Assume that the type `transformation` is defined.)

`TR-THENL` If `L` is a list of transformations, then the `(f TR-THENL L)` is a transformation that first applies the transformation `f` and then applies the transformations in `L` to the resulting incomplete leafs in a left to right order.

`TR-THEN` The transformation `(f TR-THEN f')` first applies the transformation `f` and then applies `f'` to each incomplete leaf of the result.

`TR-ORELSE` The transformation `(f TR-ORELSE f')` first tries to apply `f`. If this is successful the result is returned. If it fails the result is that of applying `f'`

In addition the system supplies the following functions:

`mark : string -> transformation`
For a string `s` and tactic tree `r`, `mark(s)(r)` evaluates to `r` and has the side effect of binding `s` to a copy of `r` in the mark/copy environment.

`copy : string -> transformation`
For a string `s` and tactic tree `r`, `copy(s)(r)` evaluates to the tactic tree `r'` associated with `s` in the mark/copy environment, and fails if no such binding exists.

`frontier : transformation -> transformation`
For a transformation `f` and a tactic tree `r`, `frontier(f)(r)` returns a tactic tree that results from applying `f` to each incomplete leaf of `r`. It fails if any application of `f` fails. Note that this could be defined as `idtrans TR-THEN f`.

These are patterned after functions with the same names from the Nuprl system. Although, as will be discussed further in Section 7.8 of the next chapter, these functions have different types in Nuprl where transformations arise as one mode of invoking tactics.

Chapter 7

Proof development with tactic trees

This chapter explores issues related to proof development in the tactic-tree framework. The chapter is organized as follows. Section 7.1 contains a discussion of formal systems and formal derivations. Section 7.2 outlines the use of tactics as a *goal-directed* means of constructing formal derivations. Emphasis is placed on the soundness and completeness of a collection of tactics with respect to a formal system. Section 7.3 discusses extending the tactic and tactic-tree frameworks with search tactics that incorporate backtracking. Section 7.4 presents new notation for tactic trees and tactics. Section 7.5 presents a definition of *refinement logics.*

Two formal systems are presented as examples: Hoare logic in Section 7.6 and a fragment of type theory in Section 7.7. Several tactic systems are developed for each formal system. The design of these tactic systems is motivated by their intended use in the construction of tactic trees. For example, one collection of tactics for Hoare logic allows for the extraction of programs from tactic trees constructed with them. The tactics defined for type-theory example give rise to tactic trees that closely resemble the refinement trees of Nuprl. In both the representation of Hoare logic and type theory, extraction of programs from a refinement tree r is accomplished in a uniform manner with the function syn(r), rather than, as in Nuprl, with an entirely separate mechanism. Our approach also clarifies the relationship between a formal system and a refinement presentation of this system.

Finally, Section 7.8 contrasts the tactic-tree approach with the the apparatus of refinement currently implemented in Nuprl and Section 7.9 concludes with a few remarks about the advantages of the tactic-tree approach.

7.1 Formal derivations

A formal presentation of a logic is a mathematically precise presentation of the *syntax*, *inference rules*, and *proofs* of the logic, that together define a formal *calculus of deduction.* Such a presentation contains various *judgment forms*, or *sentences*, J, representing assertions in the logic. Inference rules are assumed to be presented in the form

$$R \qquad \frac{J_1, \ldots, J_n}{J}$$

where R is the rule name and $J, J_1, \ldots, J_n$ are judgments. The J_i are called the *premises* of the rule, while J is called the *consequence.* Rules with no premises are called axioms. Such rules are usually presented *schematically.* That is, the judgments $J_0, J_1, \ldots, J_n$ may contain meta-variables ranging over various syntactic constructs. An instance J' of such a schematic judgment J is obtained by instantiating the meta-variables of J. Rules are often constrained by *side conditions* C^R that impose constraints on valid instances of the rule. In this case the rule will be presented as

$$R \qquad \frac{J_1, \ldots, J_n}{J} \qquad \text{provided } C^R$$

A *valid instance* of such a rule is of the form

$$\frac{J'_1, \ldots, J'_n}{J'}$$

where $J', J'_1, \ldots, J'_n$ are a consistent instantiation of $J, J_1, \ldots, J_n$ that satisfy C^R. A *formal system* is a finite collection $\mathcal{R}$ of such rules.

A set of rules $\mathcal{R}$ generates a set of *derivations.* The rule name R is treated as a term constructor and derivations as terms of the form

$$R(J, d_1, \ldots, d_k).$$

where J is a judgment form, R is a rule name, and each d_i is a formal derivation. Given a derivation $d = R(J, d_1, \ldots, d_k)$, define the judgment $\mathrm{concl(d)} = J$. Given a set of rules $\mathcal{R}$, inductively define the *set formal derivations over* $\mathcal{R}$ as follows:

Definition 39 **($\mathcal{D}^{\mathcal{R}}$)** *The set of formal derivations over $\mathcal{R}$, denoted $\mathcal{D}^{\mathcal{R}}$, is defined to be all terms d of the form*

$$R(J', d_1, \ldots, d_k),$$

where $d_1, \ldots, d_k \in \mathcal{D}^{\mathcal{R}}$, $\mathrm{concl(d_i)} = J'_i$, and

$$\frac{J'_1, \ldots, J'_k}{J'}$$

is a valid instance of the rule $R \in \mathcal{R}$.

This assumes that $\mathcal{R}$ contains at least one axiom R generating terms $R(J')$ having no subderivations. A judgment J' is *derivable* in $\mathcal{R}$ if there is a formal derivation $d \in \mathcal{D}^{\mathcal{R}}$ such that concl(d) $= J'$.

7.2 Designing tactics to encode formal derivations

This section considers the task of developing a tactic system for a given set of formal derivations $\mathcal{D}^{\mathcal{R}}$. Although any collection of tactics can be used to construct tactic trees, a collection of tactics may be *designed* for exactly this purpose. There can be no one approach that works uniformly for all logics. Such a development must necessarily involve *design decisions* influenced by the particular logic under consideration and motivated by the ultimate mode of use envisioned.

This design process begins with the choice of the goal and event sets and the achievement relation. Given a set of formal derivations $\mathcal{D}^{\mathcal{R}}$, the most obvious problem space $(\mathcal{G}, \mathcal{E}, \mathcal{A})$ is obtained by letting $\mathcal{G}$ be the set of judgment forms, $\mathcal{E}$ be the set of formal derivations, and to say that d achieves J' if and only if d is a derivation with concl(d) $= J'$.

However, many other problem spaces are possible. For example, Section 7.6 defines a set of formal derivations $\mathcal{D}^{\mathbf{HL}}$ for the well-known logic of Hoare [Hoa69]. This logic has judgments of the form $\vdash \{p\}\ S\ \{q\}$, where p and q are assertions and S is a program. One problem space is defined as having goals of the form $\vdash p \rightsquigarrow q$, which are meant to capture the problem of finding a program S such that $\vdash \{p\}\ S\ \{q\}$ is provable in Hoare logic. Events are simply judgments $\vdash \{p\}\ S\ \{q\}$ that achieve a goal $\vdash p \rightsquigarrow q$ if and only if there exists a formal derivation $d \in \mathcal{D}^{\mathbf{HL}}$ with concl(d) $= \vdash \{p\}\ S\ \{q\}$. This encoding of Hoare logic allows for the construction of tactic trees r that contain as goals only input-output specifications of the form $\vdash p \rightsquigarrow q$. A program can then be *extracted* from a complete tactic tree r by taking if from syn$(r) = \vdash \{p\}\ S\ \{q\}$. In this way, a refinement system, similar in many respects to one in Bate's thesis [Bat79], arises in a natural way from the construction of tactic trees.

Similarly, Section 7.7 introduces a type theory having judgments of the form $\Gamma \vdash a \in A$, which assert that under assumptions Γ that the term a has type A. A problem space is defined having goals[1] of the form $\Gamma \gg A$. Events are simply judgments $\Gamma \vdash a \in A$ that achieve a goal $\Gamma \gg A$ if and only if there exists a derivation d with concl(d) $= \Gamma \vdash a \in A$. If r is a complete tactic tree with root$(r) = \Gamma \gg A$ constructed over a collection of valid tactic trees for this problem space, then $\Gamma \vdash a \in A =$ syn(r), where a corresponds to the term extracted from a Nuprl refinement of $\Gamma \gg A$.

These examples should help motivate the following definition:

[1]The goal form $\Gamma \gg A$ is meant to suggest the judgment forms of Nuprl.

Definition 40 (an encoding of $\mathcal{D}^{\mathcal{R}}$)
Define an encoding *of a set of formal derivations $\mathcal{D}^{\mathcal{R}}$, to be a tuple*

$$((\mathcal{G}, \mathcal{E}, \mathcal{A}), \mathcal{I}),$$

where $(\mathcal{G}, \mathcal{E}, \mathcal{A})$ is a problem space, and $\mathcal{I}$ is a relation between judgments (sentences) of $\mathcal{D}^{\mathcal{R}}$ and event-goal pairs such that

1. *for all $g \in \mathcal{G}$ and for all $e \in \mathcal{E}$, $\mathcal{A}(e, g)$ if and only if there exists some $d \in \mathcal{D}^{\mathcal{R}}$ with $\mathcal{I}(\mathrm{concl(d)}, (e, g))$,*
2. *for all $d \in \mathcal{D}^{\mathcal{R}}$, there exists $g \in \mathcal{G}$, $e \in \mathcal{E}$ with $\mathcal{A}(e, g)$ such that $\mathcal{I}(\mathrm{concl(d)}, (e, g))$.*

Thus, the relation $\mathcal{I}$ serves to relate event-goal pairs and the derivable sentences of $\mathcal{D}^{\mathcal{R}}$. So for example, in the Hoare logic example mentioned above, the relation $\mathcal{I}$ is defines as

$$\mathcal{I}(\vdash \{p\}\ S\ \{q\}, (S', \vdash p' \leadsto q')) \text{ iff } p = p', q = q', \text{ and } S = S'.$$

Given an encoding of a system of formal derivations $\mathcal{D}^{\mathcal{R}}$, some tactic systems can construct all derivations in $\mathcal{D}^{\mathcal{R}}$:

Definition 41 ($\mathcal{B}$ is complete w.r.t. an encoding) *A set of tactics*

$$\mathcal{B} \subseteq \mathcal{VT}^{(\mathcal{G}, \mathcal{E}, \mathcal{A})}$$

is complete w.r.t. an encoding *of the set of formal derivations $\mathcal{D}^{\mathcal{R}}$,*

$$((\mathcal{G}, \mathcal{E}, \mathcal{A}), \mathcal{I}),$$

if for all $d \in \mathcal{D}^{\mathcal{R}}$, there exists some $g \in \mathcal{G}$ and some $T \in \mathcal{B}$ such that $T(g) = ((), v)$ and $\mathcal{I}(\mathrm{concl(d)}, (v(), g))$.

This definition seems natural enough, but once an encoding of a set of derivations is defined it would be better if the completeness of a set of tactics could be demonstrated with reference only to the problem space of this encoding. This motivates the following definition:

Definition 42 ($\mathcal{B}$ is strongly complete) *A set of tactics*

$$\mathcal{B} \subseteq \mathcal{VT}^{(\mathcal{G}, \mathcal{E}, \mathcal{A})}$$

is strongly complete w.r.t the problem space $(\mathcal{G}, \mathcal{E}, \mathcal{A})$ *if for every $g \in \mathcal{G}$ and every $e \in \mathcal{E}$ with $\mathcal{A}(e, g)$, there exists some $T \in \mathcal{B}$ such that $T(g) = ((), v)$ and $v() = e$.*

The term *strongly complete* is used here to distinguish this concept from another natural definition of completeness. Define a goal g to be *achievable* w.r.t. $(\mathcal{G}, \mathcal{E}, \mathcal{A})$, if there exists some $e \in \mathcal{E}$ such that $\mathcal{A}(e, g)$. Then define $\mathcal{B}$ to be *complete*, if for every achievable $g \in \mathcal{G}$, there exists some $T \in \mathcal{B}$ that achieves g, that is, such that $T(g) = ((), v)$. However, the definition of strong completeness splits into two parts the problem of demonstrating that a set of tactics is complete with respect to an encoding of a set of formal derivations:

Fact 18 *If $((\mathcal{G}, \mathcal{E}, \mathcal{A}), \mathcal{I})$, is an encoding of $\mathcal{D}^{\mathcal{R}}$, and $\mathcal{B}$ is strongly complete w.r.t. the problem space $(\mathcal{G}, \mathcal{E}, \mathcal{A})$, then $\mathcal{B}$ is complete w.r.t this encoding of $\mathcal{D}^{\mathcal{R}}$.*

In other words, the task of defining a complete set of tactics for a set of derivations $\mathcal{D}^{\mathcal{R}}$ can be carried out in two stages: first, define an encoding for $\mathcal{D}^{\mathcal{R}}$; second, define a set of tactics that is strongly complete with respect to the problem space of this encoding.

The approach, used below, to defining a collection of tactics for a set of formal derivations can be outlined in the following steps. First, as mentioned, an encoding $((\mathcal{G}, \mathcal{E}, \mathcal{A}), \mathcal{I})$ for $\mathcal{D}^{\mathcal{R}}$ is defined. Next, for each rule of $\mathcal{R}$ a collection of (parameterized) tactics is defined and justified by the rule. In this way a set of basic valid tactics $\mathcal{B}$ is defined. This collection of tactics is then used to define the tactic system $\mathcal{TS}^{(\mathcal{G},\mathcal{E},\mathcal{A})}(\mathcal{B}, \mathcal{C})$ which is then claimed to be strongly complete w.r.t. $(\mathcal{G}, \mathcal{E}, \mathcal{A})$.

7.3 Search tactics

Of course, an implementation of $\mathcal{TS}^{(\mathcal{G},\mathcal{E},\mathcal{A})}(\mathcal{B}, \mathcal{C})$ would be extended with other tactics. Of particular interest are extensions containing search tactics that provide general proof search strategies with the use of some notion of backtracking. One way of addressing this directly in the set-theoretic account is to define the set of tactics as the (partial) functions

$$\mathcal{T}^{(\mathcal{G},\mathcal{E},\mathcal{A})} = \mathcal{G} \to \left(\mathcal{U} + \bigcup_{0 \leq n} (\mathcal{G}^n \times (\mathcal{E}^n \to \mathcal{E}))\right),$$

where $\mathcal{U}$ is a *failure set*. Thus, if $g \in \mathcal{G}$ and $T \in \mathcal{T}^{\mathcal{G},\mathcal{E},\mathcal{U}}$, then either

$$T(g) = \mathrm{inl}(u)$$

for some $u \in \mathcal{U}$, or

$$T(g) = \mathrm{inr}((g_1, \ldots, g_n), v)$$

where v is an n-ary validation. In the first case the tactic T is said to *fail*, or to be *unsuccessful*, and in the second case it is said to be *successful*. A similar change could be made to the definition of tactic-tree transformations,

$$\mathcal{TR}^{\mathcal{B}} = \mathcal{TT}^{\mathcal{B}} \to (\mathcal{U} + \mathcal{TT}^{\mathcal{B}}).$$

These new definitions could then be carried through all of our definitions involving tactics and tactic trees.

However, extensions to the base system $\mathcal{TS}^{(\mathcal{G},\mathcal{E},\mathcal{A})}(\mathcal{B},\mathcal{C})$ should be nonessential (or *conservative*) in the following sense:

Definition 43 ($\mathcal{T}_2$ is an conservative extension of $\mathcal{T}_1$) *Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two sets of valid tactics. Then $\mathcal{T}_2$ is an* extension *of $\mathcal{T}_1$ if $\mathcal{T}_1 \subseteq \mathcal{T}_2$. $\mathcal{T}_2$ is a* conservative extension *of $\mathcal{T}_1$ if it is an extension of $\mathcal{T}_1$ and if for every tactic $T_2 \in \mathcal{T}_2$ and any $g \in \mathcal{G}$ with $T_2(g) = ((g_1, \ldots, g_n), v)$ there exists a tactic $T_1 \in \mathcal{T}_1$ with $T_1(g) = ((g_1, \ldots, g_n), v)$.*

Thus, any refinement of a goal g that can be obtained with a tactic from $\mathcal{T}_2$ could just as well have been obtained with a tactic from $\mathcal{T}_1$. It is clear that any conservative extension of a valid and strongly complete set of tactics is itself valid and strongly complete.

7.4 Notation for tactics and tactic trees

Given a problem space $(\mathcal{G},\mathcal{E},\mathcal{A})$, define *parameterized tactics* to be functions

$$T \in (A_1 \times A_2 \times \cdots \times A_k) \rightarrow \mathcal{T}^{(\mathcal{G},\mathcal{E},\mathcal{A})}$$

where the A_i are called *argument sets.* If $\mathcal{B}$ is a set of parameterized tactics, define Close($\mathcal{B}$), called the *closure* of $\mathcal{B}$, to be the set of tactics generated by $\mathcal{B}$. That is, if $T \in \mathcal{B}$,

$$T \in (A_{i_1} \times A_{i_2} \times \cdots \times A_{i_k}) \rightarrow \mathcal{T}^{(\mathcal{G},\mathcal{E},\mathcal{A})},$$

and $a_i \in A_i$, then $T(a_1, \ldots, a_k) \in \text{Close}(\mathcal{B})$.

To increase readability the following notation will be used when presenting tactic trees. This *display form* "hides" the validations as well as providing a form that is easier to read. The display form for leaf(g) is

g
by ??

while a tactic tree node of the form

$$\text{node}(g, T(a_1, \ldots, a_k), (r_1, \ldots, r_n), v),$$

where T is a parameterized tactic with arguments $a_1, \ldots, a_k$, is presented as

g
by T with $a_1, \ldots, a_k$
  t_1
  $\vdots$
  t_n

where each t_i is the display form of r_i.

For the purpose of illustration, a few rules from a first-order logic (taken from Dummett [Dum77]) are presented here.

$$\text{Hyp} \quad \frac{}{\Gamma, A, \Gamma' \vdash A}$$

$$\vdash\supset \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}$$

$$\supset\vdash \quad \frac{\Gamma, B \vdash C \quad \Gamma \vdash A}{\Gamma, A \supset B \vdash C}$$

Judgment forms are *sequents* of the form $\Gamma \vdash A$ where Γ is a finite list of first-order formulas. The most obvious problem space $(\mathcal{G}, \mathcal{E}, \mathcal{A})$ will be used for this example: let $\mathcal{G}$ be the set of formal derivations, $\mathcal{E}$ be the set of judgment forms, and say that d achieves J' if and only if d is a derivation with concl(d) $= J'$. For each rule

$$R \quad \frac{\Gamma_1 \vdash A_1, \ldots \Gamma_n \vdash A_n}{\Gamma \vdash A}$$

define a (parameterized) tactic R–tac$(a_1, \ldots, a_k)$ that "inverts" this rule. That is,

$$R\text{–tac}(a_1, \ldots, a_k)(\Delta \vdash B) = ((\Delta_1 \vdash B_1, \ldots \Delta_n \vdash B_n), v)$$

where $v = \lambda x_1, \ldots, x_n.R(\Delta \vdash B, (x_1, \ldots, x_n))$ and

$$\frac{\Delta_1 \vdash B_1, \ldots \Delta_n \vdash B_n}{\Delta \vdash B}$$

is a valid instance of the rule R. So, from the rules Hyp, $\vdash\supset$, and $\supset\vdash$, define the following tactics:

$$\text{Hyp–tac}(\Gamma, A, \Gamma' \vdash A) = ((), \lambda d.\text{Hyp}(\Gamma, A, \Gamma' \vdash A, ()))$$

$$\vdash\supset\text{–tac}(\Gamma \vdash A \supset B) = ((\Gamma, A \vdash B), \lambda d. \vdash\supset (\Gamma \vdash A \supset B, d))$$

$$\begin{aligned} &\supset\vdash\text{–tac}(A \supset B)(\Gamma, A \supset B \vdash C) = \\ &\quad ((\Gamma, B \vdash C, \Gamma \vdash A), \lambda d_1, d_2. \supset\vdash (\Gamma \vdash C, d_1, d_2)) \end{aligned}$$

Implicit in these definitions is *failure* in the event that arguments or goals are not of the correct form.

Figure 7.1 uses the notation described above to present several stages in a valid development of a tactic tree constructed with the tactics Hyp–tac, $\vdash\supset$–tac, and $\supset\vdash$–tac.

A display notation that is similar to the display of tactic trees defined above will also be used for *defining* tactics. A (parameterized) tactic T is presented by a schematic

$\vdash (A \supset B) \supset ((B \supset C) \supset (A \supset C))$
by ??

$\vdash (A \supset B) \supset ((B \supset C) \supset (A \supset C))$
by $\vdash\supset$−tac
  $A \supset B \vdash (B \supset C) \supset (A \supset C)$
  by $\vdash\supset$−tac
    $A, \supset B, B \supset C \vdash A \supset C$
    by $\vdash\supset$−tac
      $A \supset B, B \supset C, A \vdash C$
      by ??

$\vdash (A \supset B) \supset ((B \supset C) \supset (A \supset C))$
by $\vdash\supset$−tac
  $A \supset B \vdash (B \supset C) \supset (A \supset C)$
  by $\vdash\supset$−tac
    $A, \supset B, B \supset C \vdash A \supset C$
    by $\vdash\supset$−tac
      $A \supset B, B \supset C, A \vdash C$
      by $\supset\vdash$−tac with $A \supset B$
        $B, B \supset C, A \vdash C$
        by $\supset\vdash$−tac with $B \supset C$
          $B, C, A \vdash C$
          by Hyp−tac
          $C \vdash C$
          by Hyp−tac
        $B \supset C, A \vdash A$
        by Hyp−tac

Figure 7.1: Several stages in a valid development

$g \qquad [e]$
by T with $a_1, \ldots, a_k$
$\quad g_1 \qquad [e_1]$
$\quad \vdots \qquad \vdots$
$\quad g_n \qquad [e_n]$

Figure 7.2: Refinement style *definition* of the tactic T.

$\Gamma, A, \Gamma' \vdash A \qquad [\mathbf{Hyp}(\Gamma, A, \Gamma' \vdash A)]$
by **Hyp–tac**

$\Gamma \vdash A \supset B \qquad [\vdash\supset (\Gamma \vdash A \supset B, d_1)]$
by $\vdash\supset$–**tac**
$\quad \Gamma, A \vdash B \qquad [d_1]$

$\Gamma, A \supset B \vdash C \qquad [\supset\vdash (\Gamma, A \supset B \vdash C, d_1, d_2)]$
by $\supset\vdash$–**tac** with $A \supset B$
$\quad \Gamma, B \vdash C \qquad [d_1]$
$\quad \Gamma \vdash A \qquad [d_2]$

Figure 7.3: Defining tactics in the refinement style

diagram suggesting the node it would produce when used in a tactic tree. Such a diagram is presented in Figure 7.2 defining a a (parameterized) tactic T with k arguments. In such a diagram the expressions

$$g, g_1, \ldots, g_n, a_1, \ldots, a_k, e, e_1, \ldots, e_n$$

may contain metavariables ranging over syntactic constructs. The diagram defines the tactic T, which, for all consistently instantiated instances

$$g', g'_1, \ldots, g'_n, a'_1, \ldots, a'_k, e', e'_1, \ldots, e'_n$$

of the diagram's expressions,

$$T(a'_1, \ldots, a'_k)(g') = ((g'_1, \ldots, g'_n), v),$$

where $v(e'_1, \ldots, e'_n) = e'$. Implicit in this definition is the *failure* of T when applied to arguments of wrong form. This is called the *refinement style* of defining a tactic T.

For example, in Figure 7.3 this notation is used to *define* the tactics Hyp$-$tac, $\vdash\supset-$tac, and $\supset\vdash-$tac.

7.5 Refinement logics

Tactic trees can be viewed as a means of constructing events to achieve goals. That is, if r is a complete and valid tactic tree, then $\mathcal{A}(\mathrm{syn}(r), \mathrm{root}(r))$, and r is said to *achieve* $g = \mathrm{root}(r)$.

Definition 44 ($\mathcal{TT}^{\mathcal{B}}$ **is strongly complete)** *Given problem space* $(\mathcal{G}, \mathcal{E}, \mathcal{A})$ *and a set of tactics*

$$\mathcal{B} \subseteq \mathcal{VT}^{(\mathcal{G},\mathcal{E},\mathcal{A})},$$

the set of tactic trees $\mathcal{TT}^{\mathcal{B}}$ *is* strongly complete w.r.t the problem space $(\mathcal{G}, \mathcal{E}, \mathcal{A})$, *if for every* $g \in \mathcal{G}$ *and every* $e \in \mathcal{E}$ *with* $\mathcal{A}(e, g)$, *there exists some complete* $r \in \mathcal{TT}^{\mathcal{B}}$ *with* $\mathrm{root}(r) = g$ *and* $\mathrm{syn}(r) = e$.

Fact 19 *The set of tactics* $\mathcal{TS}^{(\mathcal{G},\mathcal{E},\mathcal{A})}(\mathcal{B}, \mathcal{C})$ *is strongly complete if and only if* $\mathcal{TT}^{\mathcal{B}}$ *is strongly complete.*

This implies that, in the context of tactic trees, only the set of tactics $\mathcal{B}$ need be presented. Furthermore, if $\mathcal{B}$ is generated by the closure of a finite set of parameterized tactics, then only these tactics need be defined. Therefore, such a collection of parameterized tactics, if presented in the refinement style, will be called a a *refinement logic.*

$$\text{Axiom}\ \frac{}{\vdash \{p[t/x]\}\ x{:=}t\ \{p\}}$$

$$\text{Seq}\ \frac{\vdash \{p\}\ S_1\ \{r\} \quad \vdash \{r\}\ S_2\ \{q\}}{\vdash \{p\}\ S_1;S_2\ \{q\}}$$

$$\text{If}\ \frac{\vdash \{p \wedge e\}\ S_1\ \{q\} \quad \vdash \{p \wedge \neg e\}\ S_2\ \{q\}}{\vdash \{p\}\ \text{if } e \text{ then } S_1 \text{ else } S_2 \text{ fi}\ \{q\}}$$

$$\text{While}\ \frac{\vdash \{p \wedge e\}\ S\ \{p\}}{\vdash \{p\}\ \text{while } e \text{ do } S \text{ od}\ \{p \wedge \neg e\}}$$

$$\text{Cons}\ \frac{\vdash p \supset p_1 \quad \vdash \{p_1\}\ S\ \{q_1\} \quad \vdash q_1 \supset q}{\vdash \{p\}\ S\ \{q\}}$$

Figure 7.4: The inference rules of **HL**.

7.6 Example 1: Hoare Logic

This section develops four approaches to encoding the well-known formal system introduced by Hoare [Hoa69] for reasoning about the correctness of programs. The first approach essentially inverts the inference rules of the logic. The other three approaches provide a means of *extracting* programs from tactic trees. Two of these approaches involve representing formal derivations directly in event sets while the other does not.

This presentation is restricted to simple *while* programs given by the grammar

$$\begin{array}{rcl} S & ::= & x{:=}exp \\ & \mid & S_1;S_2 \\ & \mid & \text{if } e \text{ then } S_1 \text{ else } S_2 \text{ fi} \\ & \mid & \text{while } e \text{ do } S \text{ od} \end{array}$$

where S ranges over while programs, x ranges over an infinite set of variables, exp ranges over some set of integer expressions, and e ranges over a set of quantifier free boolean formulas. Assume a first-order language with equality $\mathcal{L}$ as well as a a system of formal derivations $\mathcal{D}^{\mathbf{L}}$, encompassing at least Peano arithmetic, for proving judgments of the form $\Gamma \vdash p$, where $p \in \mathcal{L}$ and Γ is a list of expressions in $\mathcal{L}$. Let $\mathcal{J}^{\mathbf{L}}$ denote the set of these first-order judgments. Assume that the quantifier free formulas[2] e are a subset of $\mathcal{L}$. In addition, the logic contains judgments of the form

$$\vdash \{p\}\ S\ \{q\},$$

[2] To make this presentation completely rigorous, a distinction must be made between *logical variables* and *program variables*. This distinction will, however, be glossed over here.

where S is a while program and $p, q \in \mathcal{L}$. Informally, this asserts that if S is executed in any state satisfying p then if it terminates the resulting state will satisfy q. Let $\mathcal{J}^{\mathbf{HL}}$ denote the set of these judgments. The system **HL** consists of the rules generating $\mathcal{D}^{\mathbf{L}}$ together with the rules presented in figure 7.4.

The first encoding of **HL**,

$$((\mathcal{G}_1^{\mathbf{HL}}, \mathcal{E}_1^{\mathbf{HL}}, \mathcal{A}_1^{\mathbf{HL}}), \mathcal{I}_1^{\mathbf{HL}}),$$

is defined as follows. The problem space is made up of the set of goals

$$\mathcal{G}_1^{\mathbf{HL}} = \mathcal{J}^{\mathbf{L}} \cup \mathcal{J}^{\mathbf{HL}}$$

the set of events

$$\mathcal{E}_1^{\mathbf{HL}} = \mathcal{D}^{\mathbf{L}} \cup \mathcal{D}^{\mathbf{HL}},$$

and the achievement relation

$$\mathcal{A}_1^{\mathbf{HL}}(d, \Gamma \vdash p) \text{ iff } d \in \mathcal{D}^{\mathbf{L}} \text{ and concl(d) } = \Gamma \vdash p,$$

$$\mathcal{A}_1^{\mathbf{HL}}(d, \vdash \{p\}\ S\ \{q\}) \text{ iff } d \in \mathcal{D}^{\mathbf{HL}} \text{ and concl(d)} = \vdash \{p\}\ S\ \{q\}.$$

The relation $\mathcal{I}_1^{\mathbf{HL}}$ is defined as

$$\mathcal{I}_1^{\mathbf{HL}}(\vdash \{p\}\ S\ \{q\}, (d, \vdash \{p'\}\ S'\ \{q'\})) \text{ iff concl(d)} = \vdash \{p\}\ S\ \{q\}, p = p', q = q', S = S',$$

$$\mathcal{I}_1^{\mathbf{HL}}(\Gamma \vdash p, (d, \Gamma' \vdash p',)) \text{ iff concl(d)} = \Gamma \vdash p, \Gamma = \Gamma', \text{ and } p = p'.$$

Assume that $\mathcal{B}^{\mathbf{L}}$ is a set of tactics such that the tactic system $\mathcal{TS}(\mathcal{B}^{\mathbf{L}}, \mathcal{C})$ is strongly complete with respect to derivability in $\mathcal{D}^{\mathbf{L}}$. The collection of tactics $\mathcal{B}_1^{\mathbf{HL}}$ is $\mathcal{B}^{\mathbf{L}}$ together with the closure of the parameterized tactics defined in Figure 7.5. The rule Cons has been used to define two distinct tactics. In the definition of tactic Cons$-$tac$_1$, d_3 is a proof that $\vdash q \supset q$, and in the definition of Cons$-$tac$_2$, d_1 is a proof that $\vdash p \supset p$. Define the tactic system

$$\mathcal{TS}_1^{\mathbf{HL}} = \mathcal{TS}^{(\mathcal{G}_1^{\mathbf{HL}}, \mathcal{E}_1^{\mathbf{HL}}, \mathcal{A}_1^{\mathbf{HL}})}(\mathcal{B}_1^{\mathbf{HL}}, \mathcal{C}).$$

It is easy to verify the following claim.

Fact 20 *Each tactic in* $\mathcal{B}_1^{\mathbf{HL}}$ *is valid and* $\mathcal{TS}_1^{\mathbf{HL}}$ *is strongly complete.*

The second approach provides a means of *extracting* programs from the specification of a state transition. For $d \in \mathcal{D}^{\mathbf{HL}}$, define Prog($d$) to be the program of the conclusion of d. That is, if concl(d) $= \vdash \{p\}\ S\ \{q\}$, then Prog(d) $= S$. Define a new goal form,

$\vdash \{p[t/x]\}\ x{:=}t\ \{p\}$	$[\mathrm{Axiom}(\vdash \{p[t/x]\}\ x{:=}t\ \{p\})]$
by Axiom−tac	
$\vdash \{p\}\ S_1;S_2\ \{q\}$	$[\mathrm{Seq}(\vdash \{p\}\ S_1;S_2\ \{q\}, d_1, d_2)]$
by Seq−tac with r	
$\quad\vdash \{p\}\ S_1\ \{r\}$	$[d_1]$
$\quad\vdash \{r\}\ S_2\ \{q\}$	$[d_2]$
$\vdash \{p\}$ if e then S_1 else S_2 fi $\{q\}$	[If($\vdash \{p\}$ if e then S_1 else S_2 fi $\{q\}, d_1, d_2$)]
by If−tac	
$\quad\vdash \{p \wedge e\}\ S_1\ \{q\}$	$[d_1]$
$\quad\vdash \{p \wedge \neg e\}\ S_2\ \{q\}$	$[d_2]$
$\vdash \{p\}$ while e do S od $\{p \wedge \neg e\}$	[While($\vdash \{p\}$ while e do S od $\{p \wedge \neg e\}, d_1$)]
by While−tac	
$\quad\vdash \{p \wedge e\}\ S\ \{p\}$	$[d_1]$
$\vdash \{p\}\ S\ \{q\}$	$[\mathrm{Cons}(\vdash \{p\}\ S\ \{q\}, d_1, d_2, d_3)]$
by Cons−tac$_1$ with r	
$\quad\vdash p \supset r$	$[d_1]$
$\quad\vdash \{r\}\ S\ \{q\}$	$[d_2]$
$\vdash \{p\}\ S\ \{q\}$	$[\mathrm{Cons}(\vdash \{p\}\ S\ \{q\}, d_1, d_2, d_3)]$
by Cons−tac$_2$ with r	
$\quad\vdash r \supset q$	$[d_2]$
$\quad\vdash \{p\}\ S\ \{r\}$	$[d_3]$

Figure 7.5: The tactics generating $\mathcal{B}_1^{\mathbf{HL}}$

$\vdash p \leadsto q$, where $p, q \in \mathcal{L}$, to represent the problem of finding a a program S and a derivation $d \in \mathcal{D}^{\mathbf{HL}}$ such that concl(d) $=\vdash \{p\}\ S\ \{q\}$. Define the encoding

$$((\mathcal{G}_2^{\mathbf{HL}}, \mathcal{E}_2^{\mathbf{HL}}, \mathcal{A}_2^{\mathbf{HL}}), \mathcal{I}_2^{\mathbf{HL}}),$$

where the set of goals is defined as

$$\mathcal{G}_2^{\mathbf{HL}} = \{\vdash p \leadsto q \mid p, q \in \mathcal{L}\} \cup \mathcal{J}^{\mathbf{L}},$$

the set of events is defined as

$$\mathcal{E}_2^{\mathbf{HL}} = \mathcal{D}^{\mathbf{L}} \cup \mathcal{D}^{\mathbf{HL}},$$

and the achievement relation is defined as

$$\mathcal{A}_2^{\mathbf{HL}}(d, \Gamma \vdash p) \text{ iff } \mathcal{A}_1^{\mathbf{HL}}(d, \Gamma \vdash p),$$

$$\mathcal{A}_2^{\mathbf{HL}}(d, \vdash p \leadsto q) \text{ iff concl(d)} = \vdash \{p\}\ S\ \{q\} \text{ for some program } S.$$

The relation $\mathcal{I}_1^{\mathbf{HL}}$ is defined as

$$\mathcal{I}_2^{\mathbf{HL}}(\Gamma \vdash p, (d, \Gamma' \vdash p,)) \text{ iff } \mathcal{I}_1^{\mathbf{HL}}(\Gamma \vdash p, (d, \Gamma' \vdash p,)),$$

$$\mathcal{I}_2^{\mathbf{HL}}(\vdash \{p\}\ S\ \{q\}, (d, \vdash p' \leadsto q')) \text{ iff concl(d)} = \vdash \{p\}\ S\ \{q\}, p = p', \text{ and } q = q'.$$

A tactic system can now be defined that allows the *extraction* of programs from complete tactic trees with root goal of the form $\vdash p \leadsto q$. The set of basic $\mathcal{B}_2^{\mathbf{HL}}$ is defined to be $\mathcal{B}^{\mathbf{L}}$ together with the closure of the parameterized tactics defined in figure 7.6. In this figure, the notation for subgoals

$$\vdash p \leadsto q \qquad [d \text{ for } S]$$

means that d is a derivation with concl(d) $= \vdash \{p\}\ S\ \{q\}$. Again, in the definition of tactic Cons$-$tac$_3$, d_3 is a proof that $\vdash q \supset q$, and in the definition of Cons$-$tac$_4$, d_1 is a proof that $\vdash p \supset p$.

Define the tactic system

$$\mathcal{TS}_2^{\mathbf{HL}} = \mathcal{TS}^{(\mathcal{G}_2^{\mathbf{HL}}, \mathcal{E}_2^{\mathbf{HL}}, \mathcal{A}_2^{\mathbf{HL}})}(\mathcal{B}_2^{\mathbf{HL}}, \mathcal{C}).$$

Again, it is easy to verify the following claim.

Fact 21 *Each tactic in $\mathcal{B}_2^{\mathbf{HL}}$ is valid and $\mathcal{TS}_2^{\mathbf{HL}}$ is strongly complete.*

$\vdash p[t/x] \leadsto p$	[Axiom($\vdash \{p[t/x]\}\ x{:=}t\ \{p\}$)]
by Axiom$-$tac$_2$ with t for x	
$\vdash p \leadsto q$	[Seq($\vdash \{p\}\ S_1;S_2\ \{q\}, d_1, d_2$)]
by Seq$-$tac$_2$ with r	
$\quad \vdash p \leadsto r$	[d_1 for S_1]
$\quad \vdash r \leadsto q$	[d_2 for S_2]
$\vdash p \leadsto q$	[If($\vdash \{p\}$ if e then S_1 else S_2 fi $\{q\}, d_1, d_2$)]
by If$-$tac$_2$ with e	
$\quad \vdash p \wedge e \leadsto q$	[d_1 for S_1]
$\quad \vdash p \wedge \neg e \leadsto q$	[d_2 for S_2]
$\vdash p \leadsto p \wedge \neg e$	[While($\vdash \{p\}$ while e do S od $\{p \wedge \neg e\}, d_1$)]
by While$-$tac$_2$	
$\quad \vdash p \wedge e \leadsto p$	[d_1 for S_1]
$\vdash p \leadsto q$	[Cons($\vdash \{p\}\ S\ \{q\}, d_1, d_2, d_3$)]
by Cons$-$tac$_3$ with r	
$\quad \vdash p \supset r$	[d_1]
$\quad \vdash r \leadsto q$	[d_2 for S]
$\vdash p \leadsto q$	[Cons($\vdash \{p\}\ S\ \{q\}, d_1, d_2, d_3$)]
by Cons$-$tac$_4$ with r	
$\quad \vdash r \supset q$	[d_2]
$\quad \vdash p \leadsto r$	[d_3 for S]

Figure 7.6: Tactics generating $\mathcal{B}_2^{\mathbf{HL}}$

$\vdash 0 \leq x \wedge 0 \leq y \rightsquigarrow a \times y + b = x \wedge 0 \leq b \wedge \neg(y \leq b)$
by Seq$-$tac$_2$ with $a \times y + b = x \wedge 0 \leq b$
  $\vdash 0 \leq x \wedge 0 \leq y \rightsquigarrow a \times y + b = x \wedge 0 \leq b$
  by Cons$-$tac$_3$ with $0 \times y + x = x \wedge 0 \leq x$
    $\vdash 0 \leq x \wedge 0 \leq y \supset 0 \times y + x = x \wedge 0 \leq x$
      r_1
    $\vdash 0 \times y + x = x \wedge 0 \leq x \rightsquigarrow a \times y + b = x \wedge 0 \leq b$
    by Seq$-$tac$_2$ with $a \times y + x = x \wedge 0 \leq x$
      $\vdash 0 \times y + x = x \wedge 0 \leq x \rightsquigarrow a \times y + x = x \wedge 0 \leq x$
      by Axiom$-$tac$_2$ with 0 for a
      $\vdash a \times y + x = x \wedge 0 \leq x \rightsquigarrow a \times y + b = x \wedge 0 \leq b$
      by Axiom$-$tac$_2$ with x for b
  $\vdash a \times y + b = x \wedge 0 \leq b \rightsquigarrow a \times y + b = x \wedge 0 \leq b \wedge \neg(y \leq b)$
  by While$-$tac$_2$
    $\vdash a \times y + b = x \wedge 0 \leq b \wedge y \leq b \rightsquigarrow a \times y + b = x \wedge 0 \leq b$
    by Cons$-$tac$_3$ with $(a+1) \times y + (b-y) = x \wedge 0 \leq b - y$
      $\vdash a \times y + b = x \wedge 0 \leq b \wedge y \leq b \supset (a+1) \times y + (b-y) = x \wedge 0 \leq b - y$
        r_2
      $\vdash (a+1) \times y + (b-y) = x \wedge 0 \leq b - y \rightsquigarrow a \times y + b = x \wedge 0 \leq b$
      by Seq$-$tac$_2$ with $(a+1) \times y + b = x \wedge 0 \leq b$
        $\vdash (a+1) \times y + (b-y) = x \wedge 0 \leq b - y \rightsquigarrow (a+1) \times y + b = x \wedge 0 \leq b$
        by Axiom$-$tac$_2$ with $b - y$ for b
        $\vdash (a+1) \times y + b = x \wedge 0 \leq b \rightsquigarrow a \times y + b = x \wedge 0 \leq b$
        by Axiom$-$tac$_2$ with $a + 1$ for a

Figure 7.7: A tactic tree constructed over $\mathcal{B}_2^{\mathbf{HL}}$

Thus, if r is a complete tactic tree over $\mathcal{TS}_2^{\mathbf{HL}}$ with root$(r) = \vdash p \leadsto q$, then syn$(r) = d$, where d is a **HL** derivation for $\vdash \{p\}\ S\ \{q\}$. In other words, programs can be *extracted* from a tactic tree whose goals are program specifications. As an example, figure 7.7 presents a tactic tree r with root

$$\vdash 0 \leq x \wedge 0 \leq y \leadsto a \times y + b = x \wedge 0 \leq b \wedge \neg(y \leq b).$$

This goal can be read as the specification of an integer division algorithm[3]. The tactic trees r_1 and r_2 are assumed to be constructed over tactics derived from the rules generating $\mathcal{D}^{\mathcal{L}}$. It is not too hard to check that syn$(r) = d$ where

$$\text{Prog}(d) = a{:=}0;b{:=}x;\text{while } y \leq b \text{ do } b{:=}b - y;a{:=}a + 1 \text{ od}$$

and that d is a derivation for

$$\vdash \{0 \leq x \wedge 0 \leq y\}\ \text{Prog}(d)\ \{a \times y + b = x \wedge 0 \leq b \wedge \neg(y \leq b)\}.$$

A third encoding is defined by joining together the first and second approaches, in the obvious way, and adding the following tactic to serve as a bridge between the two encodings.

$\vdash p \leadsto q$ $\qquad\qquad$ $[d]$
by explicit−tac with S
$\quad\vdash \{p\}\ S\ \{q\}$ $\qquad$ $[d]$

Such a tactic system would allow for the construction of tactic trees that contain goals of the form $\vdash \{p\}\ S\ \{q\}$ and $\vdash p \leadsto q$.

The encodings developed thus far have represented derivations directly in the event sets. This may or may not be desirable if these encodings are to be used to guide the implementation of a tactic system for Hoare's logic. This motivates yet another encoding in which the event set does not contain the set of derivations. An implementation based on this encoding would not require an implementation of $\mathcal{D}^{\mathbf{HL}}$. Only the rough sketch of this encoding will be presented here, leaving the details to the interested reader.

Define the encoding

$$((\mathcal{G}_4^{\mathbf{HL}}, \mathcal{E}_4^{\mathbf{HL}}, \mathcal{A}_4^{\mathbf{HL}}), \mathcal{I}_4^{\mathbf{HL}}),$$

where the set of goals is defined as $\mathcal{G}_4^{\mathbf{HL}} = \mathcal{G}_3^{\mathbf{HL}}$, the set of events is defined as

$$\mathcal{E}_4^{\mathbf{HL}} = \mathcal{J}^{\mathbf{L}} \cup \mathcal{J}^{\mathbf{HL}},$$

[3]This proof is based on a proof given in [Apt81] that was presented in the more conventional natural deduction style.

and the achievement relation is defined as

$$\mathcal{A}_4^{\mathbf{HL}}(\Gamma \vdash p, \Gamma \vdash p) \text{ iff there exits } d \in \mathcal{D}^{\mathbf{L}}, \text{concl(d)} = \Gamma \vdash p,$$

$$\mathcal{A}_4^{\mathbf{HL}}(\vdash \{p\}\ S\ \{q\}, \vdash \{p\}\ S\ \{q\}) \text{ iff there exits } d \in \mathcal{D}^{\mathbf{HL}}, \text{concl(d)} = \vdash \{p\}\ S\ \{q\},$$

$$\mathcal{A}_4^{\mathbf{HL}}(\vdash \{p\}\ S\ \{q\}, \vdash p \rightsquigarrow q) \text{ iff there exits } d \in \mathcal{D}^{\mathbf{HL}}, \text{concl(d)} = \vdash \{p\}\ S\ \{q\}$$

Note that $\mathcal{A}_4^{\mathbf{HL}}$ is not decidable. However, a decidable consistency relation[4] $\mathcal{K}_4^{\mathbf{HL}}$ for the problem space $(\mathcal{G}_4^{\mathbf{HL}}, \mathcal{E}_4^{\mathbf{HL}}, \mathcal{A}_4^{\mathbf{HL}})$ can be defined as

$$\mathcal{K}_4^{\mathbf{HL}}(\Gamma \vdash p, \Gamma', \vdash p') \text{ iff } \Gamma = \Gamma' \text{ and } p = p',$$

$$\mathcal{K}_4^{\mathbf{HL}}(\vdash \{p\}\ S\ \{q\}, \vdash \{p'\}\ S'\ \{q'\}) \text{ iff } S = S', p = p', \text{ and } q = q',$$

$$\mathcal{K}_4^{\mathbf{HL}}(\vdash \{p\}\ S\ \{q\}, \vdash p' \rightsquigarrow q') \text{ iff } p = p' \text{ and } q = q'.$$

Then, for an implementation of a collection of almost valid tactics for this problem space, an implementation of $\mathcal{K}_4^{\mathbf{HL}}$ could be used to check achievement, as was discussed in Secion 6.3.

7.7 Example 2: A type theory

The purpose of this section is to demonstrate that the refinement rules and extraction of Nuprl can be modeled in the tactic tree framework. No attempt is made to address the entire Nuprl logic, but only a fragment large enough to make the development a convincing one. A formal system **TT** is presented that is closely related to a fragment of Martin-Löf's intuitionistic type theory [ML82]. The reader is assumed to be familiar with this theory. For a good introduction the Sambin's notes of the Padova lectures [ML] are highly recommended. The system **TT** differs from those of Martin-Löf in several respects. These differences and possible semantic or proof-theoretic justifications for **TT** will not be discussed. The only justification offered here is that from this formal system a collection of tactics can be defined so that tactic trees constructed over them correspond very closely to Nuprl refinement trees.

Four tactic systems based on **TT** will be outlined below that differ in the definition of events, goals, and achievement. As in the Hoare logic example, the first approach essentially inverts the inference rules of the logic. The other three approaches provide a means of *extracting* terms from tactic trees and two of these approaches involve representing formal derivations directly in event sets while the other does not.

Assertions of **TT** have the form

$$\Gamma \vdash a = b \in A$$

[4]See Section 6.3 for the definition of a consistency relation.

where Γ is a *context* of the form

$$x_1 \in A_1, \ldots, x_n \in A_n$$

for some $n \geq 0$, and a,b, A, $A_1, \ldots, A_n$ are terms. The variables $x_1, \ldots, x_n$ are pairwise distinct, and all free variables of the term A_{i+1} are among the $x_1, \ldots, x_i$. Terms are taken to be given by the syntax presented in Figure 5.1 of Chapter 5. The abbreviation $\Gamma \vdash a \in A$ is used to represent $\Gamma \vdash a = a \in A$. Informally, $\Gamma \vdash a = b \in A$ asserts that under the assumptions of Γ, a and b are equal elements of type A. Note that in the rules presented below distinct occurrences of the same metavariable, such as A, can be instantiated with distinct members of an α-equivalence class of terms.

The system **TT**:

Axioms:

$$\text{Hyp} \quad \frac{}{\Gamma, x \in A, \Gamma' \vdash x \in A}$$

$$\text{Thin} \quad \frac{\Gamma \vdash a = b \in A}{\Gamma, \Gamma' \vdash a = b \in A}$$

Equality rules:

$$\text{Sym} \quad \frac{\Gamma \vdash a = b \in A}{\Gamma \vdash b = a \in A}$$

$$\text{Trans} \quad \frac{\Gamma \vdash a = b \in A \quad \Gamma \vdash b = c \in A}{\Gamma \vdash a = c \in A}$$

$$\text{Sub} \quad \frac{\Gamma \vdash a = b \in A \quad \Gamma, x \in A \vdash c = d \in B}{\Gamma \vdash c[a/x] = d[b/x] \in B[a/x]}$$

The + rules:

$$\textbf{+-F} \quad \frac{\Gamma \vdash A \in \mathrm{U}_i \quad \Gamma \vdash B \in \mathrm{U}_i}{\Gamma \vdash A + B}$$

$$\textbf{+-I}_1 \quad \frac{\Gamma \vdash a \in A \quad \Gamma \vdash B \in \mathrm{U}_i}{\Gamma \vdash \mathrm{inl}(a) \in A + B}$$

$$\textbf{+-I}_2 \quad \frac{\Gamma \vdash b \in B \quad \Gamma \vdash A \in \mathrm{U}_i}{\Gamma \vdash \mathrm{inr}(b) \in A + B}$$

$$\textbf{+-E}_1 \quad \frac{\begin{array}{l}\Gamma \vdash e \in A + B \\ \Gamma, x \in A \vdash t_1 \in C[\mathrm{inl}(x)/z] \\ \Gamma, y \in B \vdash t_2 \in C[\mathrm{inr}(y)/z]\end{array}}{\Gamma \vdash \mathrm{case}(e; x.t_1; y.t_2) \in C[e/z]}$$

$$+\text{-}\mathbf{E}_2 \quad \frac{\Gamma, z \in A + B, \Gamma', x \in A \vdash t_1 \in C[\mathrm{inl}(x)/z] \qquad \Gamma, z \in A + B, \Gamma', y \in B \vdash t_2 \in C[\mathrm{inr}(y)/z]}{\Gamma, z \in A + B, \Gamma' \vdash \mathrm{case}(z; x.t_1; y.t_2) \in C}$$

$$+\text{-}\mathbf{Eq}_1 \quad \frac{\Gamma \vdash t_1[a/x] = t \in C}{\Gamma \vdash \mathrm{case}(\mathrm{inl}(a); x.t_1; y.t_2) = t \in C}$$

$$+\text{-}\mathbf{Eq}_2 \quad \frac{\Gamma \vdash t_2[b/y] = t \in C}{\Gamma \vdash \mathrm{case}(\mathrm{inr}(b); x.t_1; y.t_2) = t \in C}$$

The $\wedge$ rules :

$$\wedge\text{-}\mathbf{F} \quad \frac{\Gamma \vdash A \in \mathrm{U}_i \quad \Gamma \vdash B \in \mathrm{U}_i}{\Gamma \vdash A \wedge B \in \mathrm{U}_i}$$

$$\wedge\text{-}\mathbf{I} \quad \frac{\Gamma \vdash a \in A \quad \Gamma \vdash b \in B}{\Gamma \vdash \langle a, b \rangle \in A \wedge B}$$

$$\wedge\text{-}\mathbf{E}_1 \quad \frac{\Gamma, x \in A, y \in B \vdash b \in C[\langle x, y \rangle/z] \quad \Gamma \vdash c \in A \wedge B}{\Gamma \vdash \mathrm{spread}(c; x, y.b) \in C[c/z]}$$

$$\wedge\text{-}\mathbf{E}_2 \quad \frac{\Gamma, z \in A \wedge B, \Gamma', u \in A, v \in B \vdash t \in C[\langle u, v \rangle/z]}{\Gamma, z \in A \wedge B, \Gamma' \vdash \mathrm{spread}(z; u, v.t) \in C}$$

$$\Sigma\text{-}\mathbf{eq} \quad \frac{\Gamma \vdash d[a/x, b/y] = t \in C}{\Gamma \vdash \mathrm{spread}(\langle a, b \rangle; x, y.d) = t \in C}$$

The $\rightarrow$ rules:

$$\rightarrow\text{-}\mathbf{F} \quad \frac{\Gamma \vdash A \in \mathrm{U}_i \quad \Gamma \vdash B \in \mathrm{U}_i}{\Gamma \vdash A \rightarrow B \in \mathrm{U}_i}$$

$$\rightarrow\text{-}\mathbf{I} \quad \frac{\Gamma \vdash A \in \mathrm{U}_i \quad \Gamma, x \in A \vdash b \in B}{\Gamma \vdash \lambda x.b \in A \rightarrow B}$$

$$\rightarrow\text{-}\mathbf{E}_1 \quad \frac{\Gamma \vdash f \in A \rightarrow B \quad \Gamma \vdash a \in A}{\Gamma \vdash f(a) \in B}$$

$$\rightarrow\text{-}\mathbf{E}_2 \quad \frac{\Gamma, f \in A \rightarrow B, \Gamma' \vdash a \in A \qquad \Gamma, f \in A \rightarrow B, \Gamma', y \in B \vdash t \in C}{\Gamma, f \in A \rightarrow B, \Gamma' \vdash t[f(a)/y] \in C}$$

$$\beta\text{-}\mathbf{eq} \quad \frac{\Gamma \vdash b[a/x] = t \in C}{\Gamma \vdash (\lambda x.b)(a) = t \in C}$$

The U rules:

U-F $$\frac{}{\Gamma \vdash \mathrm{U}_j \in \mathrm{U}_i} \qquad \text{provided } j < i$$

U-C $$\frac{\Gamma \vdash T \in \mathrm{U}_j}{\Gamma \vdash T \in \mathrm{U}_i} \qquad \text{provided } j < i$$

Finally, the I rules:

I-F $$\frac{\Gamma \vdash A \in \mathrm{U}_i \quad \Gamma \vdash a \in A \quad \Gamma \vdash b \in A}{\Gamma \vdash \mathrm{I}(a, b, A) \in \mathrm{U}_i}$$

I-I $$\frac{\Gamma \vdash a = b \in A}{\Gamma \vdash \mathrm{r} \in \mathrm{I}(a, b, A)}$$

I-E $$\frac{\Gamma \vdash \mathrm{r} \in \mathrm{I}(a, b, A)}{\Gamma \vdash a = b \in A}$$

Derivations $d \in \mathcal{D}^{\mathbf{TT}}$ with concl(d) $=\ \vdash a = b \in A$ will be called *theorems.*

In the first approach to encoding **TT** with a collection of tactics is carried out much like the first tactic system development for **HL**. The set of goals $\mathcal{G}_1^{\mathbf{TT}}$ is defined to be the set of sequents $\Gamma \vdash a = b \in A$, and the set of events $\mathcal{E}_1^{\mathbf{TT}}$ to be the set of derivations $\mathcal{D}^{\mathbf{TT}}$. The achievement relation $\mathcal{A}_1^{\mathbf{TT}}(d, \Gamma \vdash a = b \in A)$ holds if and only if d is a derivation for $\Gamma \vdash a = b \in A$. Then, for each rule in **TT**

$$\frac{\Gamma_1 \vdash a_1 = b_1 \in A_1, \cdots, \Gamma_k \vdash a_k = b_k \in A_k}{\Gamma \vdash a = b \in A}$$

it must be decided what arguments are needed, given a fully instantiated consequent, to ensure that the premises can be fully instantiated. Of course, any rule can give rise to more than one basic tactic. Figure 7.8 presents only the definition of the basic tactics corresponding to the $\rightarrow$ rules of **TT**. Tactics corresponding to the other rules will not be defined here. It should be clear that (parameterized) tactics can be defined so that their closure $\mathcal{B}_1^{\mathbf{TT}}$ can be used to define the tactic system

$$\mathcal{TS}_1^{\mathbf{TT}} = \mathcal{TS}^{(\mathcal{G}_1^{\mathbf{TT}}, \mathcal{E}_1^{\mathbf{TT}}, \mathcal{A}_1^{\mathbf{TT}})}(\mathcal{B}_1^{\mathbf{TT}}, \mathcal{C})$$

so that it is strongly complete.

The second approach is an extension of the first and provides a model of *extraction.* For $d \in \mathcal{D}^{\mathbf{TT}}$, define the function TermOf so that if concl(d) $= \Gamma \vdash a = b \in A$, then TermOf$(d) = a$. Define a new goal form, $\Gamma \gg A$, to represent the problem of finding a term a and a derivation $d \in \mathcal{D}^{\mathbf{TT}}$ such that concl(d) $= \Gamma \vdash a \in A$. Define the encoding

$$((\mathcal{G}_2^{\mathbf{TT}}, \mathcal{E}_2^{\mathbf{TT}}, \mathcal{A}_2^{\mathbf{TT}}), \mathcal{I}_2^{\mathbf{TT}}),$$

$\Gamma \vdash A \rightarrow B \in U_i$	$[\rightarrow\text{-}\mathbf{F}(\Gamma \vdash a \rightarrow B \in U_i, d_1, d_2)]$
by $\rightarrow\text{-}\mathbf{F}$$-\text{tac}_1$	
$\quad\Gamma \vdash A \in U_i$	$[d_1]$
$\quad\Gamma \vdash B \in U_i$	$[d_2]$
$\Gamma \vdash \lambda x.b \in A \rightarrow B$	$[\rightarrow\text{-}\mathbf{I}(\Gamma \vdash \lambda x.b \in A \rightarrow B, d_1, d_2)]$
by $\rightarrow\text{-}\mathbf{I}$$-\text{tac}_1$	
$\quad\Gamma, x \in A \vdash b \in B$	$[d_1]$
$\quad\Gamma \vdash A \in U_i$	$[d_2]$
$\Gamma \vdash f(a) \in B$	$[\rightarrow\text{-}\mathbf{E}_1(\Gamma \vdash f(a) \in B, d_1, d_2)]$
by $\rightarrow\text{-}\mathbf{E}_1$$-\text{tac}_1$ with A	
$\quad\Gamma \vdash a \in A$	$[d_1]$
$\quad\Gamma \vdash f \in a \rightarrow B$	$[d_2]$
$\Gamma, f \in A \rightarrow B, \Gamma' \vdash t[f(a)/y] \in C$	$[\rightarrow\text{-}\mathbf{E}_2(\Gamma, f \in A \rightarrow B, \Gamma' \vdash t[f(a)/y] \in C, d_1, d_2)]$
by $\rightarrow\text{-}\mathbf{E}_2$$-\text{tac}_1$ with f on a	
$\quad\Gamma, f \in A \rightarrow B, \Gamma' \vdash a \in A$	$[d_1]$
$\quad\Gamma, f \in A \rightarrow B, \Gamma', y \in B \vdash t \in C$	$[d_2]$
$\Gamma \vdash (\lambda x.b)(a) = t \in C$	$[\beta\text{-}\mathbf{eq}(\Gamma \vdash (\lambda x.b)(a) = t \in C, d)]$
by β-**eq**$-\text{tac}_1$	
$\quad\Gamma \vdash b[a/x] = t \in C$	$[d]$

Figure 7.8: A few of the tactics generating $\mathcal{B}_1^{\mathbf{TT}}$

$\Gamma \gg A + B$ $\quad$ $[+\text{-}\mathbf{I}_1(\Gamma \vdash \mathrm{inl}(a) \in A + B, d_1, d_2)]$
by $+\text{-}\mathbf{I}_1-\mathrm{tac}_2$ at U_i
$\quad\Gamma \gg A$ $\quad$ $[d_1 \text{ for } a]$
$\quad\Gamma \vdash B \in \mathrm{U}_i$ $\quad$ $[d_2]$

$\Gamma \gg A + B$ $\quad$ $[+\text{-}\mathbf{I}_2(\Gamma \vdash \mathrm{inr}(b) \in A + B, d_1, d_2)]$
by $+\text{-}\mathbf{I}_2-\mathrm{tac}_2$ at U_i
$\quad\Gamma \gg B$ $\quad$ $[d_1 \text{ for } b]$
$\quad\Gamma \vdash A \in \mathrm{U}_i$ $\quad$ $[d_2]$

$\Gamma, z \in A + B, \Gamma' \gg C$ $\quad$ $[d]$
by $+\text{-}\mathbf{E}_2-\mathrm{tac}_2$ with z
$\quad\Gamma, z \in A + B, \Gamma', x \in A \gg C[\mathrm{inl}(x)/z]$ $\quad$ $[d_1 \text{ for } t_1]$
$\quad\Gamma, z \in A + B, \Gamma', y \in B \gg C[\mathrm{inr}(y)/z]$ $\quad$ $[d_2 \text{ for } t_2]$
where $d = +\text{-}\mathbf{E}_2(\Gamma, z \in A + B, \Gamma' \vdash \mathrm{case}(z; x.t_1; y.t_2) \in C, d_1, d_2)$

Figure 7.9: Tactics for union rules.

where a goal $g \in \mathcal{G}_2^{\mathbf{TT}}$ is either of the form $\Gamma \vdash a = b \in A$ or of the form $\Gamma \gg A$, events are derivations in $\mathcal{D}^{\mathbf{TT}}$, and the achievement relation is defined as

$$\mathcal{A}_2^{\mathbf{TT}}(d, \Gamma \vdash a = b \in A) \text{ if and only if } \mathcal{A}_1^{\mathbf{TT}}(d, \Gamma \vdash a = b \in A)$$

$$\mathcal{A}_2^{\mathbf{TT}}(d, \Gamma \gg A) \text{ iff } \mathrm{concl(d)} = \Gamma \vdash a \in A \text{ for some term } a.$$

The relation $\mathcal{I}_2^{\mathbf{TT}}$ is defined in the obvious way.

The idea is to define the tactics so that if r is a complete tactic tree with $\mathrm{root}(r) = \Gamma \gg A$, then $\mathrm{TermOf}(\mathrm{syn}(r)) = a$, which corresponds to the *extracted term* of Nuprl. For some rules R with consequence $\Gamma \vdash a \in A$, a tactic $R-\mathrm{tac}_2$ is defined on goals of the form $\Gamma \gg A$. For example, define $\mathrm{Hyp}-\mathrm{tac}_2$ as

$\Gamma, x \in A, \Gamma' \gg A$ $\quad$ $[\mathrm{Hyp}(\Gamma, x \in A, \Gamma' \vdash x \in A)]$
by $\mathrm{Hyp}-\mathrm{tac}_2$

Additional tactics are presented in Figure 7.9 and Figure 7.10. In these figures, the notation for subgoals

$$\Gamma \gg A \qquad [d \text{ for } a]$$

means that d is a derivation with $\mathrm{concl(d)} = \Gamma \vdash a \in A$. Tactics corresponding to all the rules will not be defined here. It should be clear that a set of (parameterized) tactics can be defined in this way so that their closure taken together with $\mathcal{B}_1^{\mathbf{TT}}$ defines

$\Gamma \gg A \rightarrow B$ $\quad [\rightarrow\text{-}\mathbf{I}(\Gamma \vdash \lambda x.b \in A \rightarrow B, d_1, d_2)]$
by $\rightarrow$ -**I**–tac$_2$ with U_i
$\quad \Gamma, x \in A \gg B$ $\quad [d_1 \text{ for } b]$
$\quad \Gamma \vdash A \in U_i$ $\quad [d_2]$

$\Gamma, f \in A \rightarrow B, \Gamma' \gg C$ $\quad [d]$
by $\rightarrow$ -**E**$_2$–tac$_2$ with f
$\quad \Gamma, f \in A \rightarrow B, \Gamma' \gg A$ $\quad [d_1 \text{ for } a]$
$\quad \Gamma, f \in A \rightarrow B, \Gamma', y \in B \gg C$ $\quad [d_2 \text{ for } t]$
where $d = \rightarrow\text{-}\mathbf{E}_2(\Gamma, f \in A \rightarrow B, \Gamma' \vdash t[f(a)/y] \in C, d_1, d_2)$

$\Gamma \gg A \wedge B$ $\quad [\wedge\text{-}\mathbf{I}(\Gamma \vdash \langle a, b\rangle \in A \wedge B, d_1, d_2)]$
by $\wedge$-**I**–tac$_2$
$\quad \Gamma \gg A$ $\quad [d_1 \text{ for } a]$
$\quad \Gamma \gg B$ $\quad [d_2 \text{ for } b]$

$\Gamma, z \in A \wedge B, \Gamma' \gg C$ $\quad [d]$
by $\wedge$-**E**$_2$–tac$_2$ with z
$\quad \Gamma, z \in A \wedge B, \Gamma', u \in A, v \in B \gg C[\langle u, v\rangle/z]$ $\quad [d_1 \text{ for } t]$
where $d = \wedge\text{-}\mathbf{E}_2(\Gamma, z \in A \wedge B, \Gamma' \vdash \text{spread}(z; u, v.t) \in C, d_1)$

Figure 7.10: Tactics for $\rightarrow$ and $\wedge$ rules.

a collection of tactics $\mathcal{B}_2^{\mathbf{TT}}$, which in turn defines a the strongly complete tactic system

$$\mathcal{TS}_2^{\mathbf{TT}} = \mathcal{TS}^{(\mathcal{G}_2^{\mathbf{TT}},\mathcal{E}_2^{\mathbf{TT}},\mathcal{A}_2^{\mathbf{TT}})}(\mathcal{B}_2^{\mathbf{TT}},\mathcal{C}).$$

With this done, tactic trees over $\mathcal{TS}_2^{\mathbf{TT}}$ model extraction in Nuprl in the following sense.

Fact 22
If r is a complete tactic tree over any conservative extension of $\mathcal{TS}_2^{\mathbf{TT}}$, $\mathrm{root}(r) = \Gamma \gg A$, *and* $\mathrm{syn}(r) = d$, *then* $d \in \mathcal{D}^{\mathbf{TT}}$ *with* $\mathrm{concl(d)} = \Gamma \vdash \mathrm{TermOf}(d) \in A$.

As an example, a complete tactic tree r is presented in Figure 7.11. This assumes that X, Y, and Z are closed terms, and that $r_1, \ldots r_5$ represent complete tactic trees. It is not too difficult to check that $\mathrm{syn}(r) = d$ where

$$\mathrm{TermOf}(d) = \lambda x.\langle \lambda y.x(\mathrm{inl}(y)), \lambda y.x(\mathrm{inr}(y))\rangle$$

and d is a **TT** derivation of

$$\vdash \mathrm{TermOf}(d) \in ((X+Y) \to Z) \to (X \to Z) \wedge (Y \to Z).$$

It is easy to see that the tactics defined in Figures 7.8–7.10 closely correspond to Nuprl refinement rules described in Chapter 8 of [Con86]. However, several differences are immediately apparent. First, Nuprl has *only one* kind of goal of the form $\Gamma \gg A$. Where we have a goal of the form $\Gamma \vdash a = b \in A$, Nuprl has the goal of the form $\Gamma \gg \mathrm{I}(a,b,A)$. In Nuprl, $\mathrm{I}(a,b,A)$ is written as "$a = b$ in A" and $\mathrm{I}(a,A)$, which stands for $\mathrm{I}(a,a,A)$, is written as "a in A." The term r is extracted from a goal of the form $\Gamma \gg \mathrm{I}(a,b,A)$. Second, Nuprl extraction produces only a term and not a derivation. So for example, where we have the tactic

$\Gamma \gg A \to B$	$[\to\text{-}\mathbf{I}(\Gamma \vdash \lambda x.b \in A \to B, d_1, d_2)]$
by $\to$ **-I**$-$tac$_2$ with U_i	
$\Gamma, x \in A \gg B$	$[d_1$ for $b]$
$\Gamma \vdash A \in \mathrm{U}_i$	$[d_2]$

Nuprl has the *basic refinement rule*

$\Gamma \gg A \to B$	$[\lambda x.b]$
by $\to$ **-I** with U_i	
$\Gamma, x \in A \gg B$	$[b]$
$\Gamma \gg \mathrm{I}(A, \mathrm{U}_i)$	[r]

$$
\begin{array}{l}
\gg ((X+Y) \rightarrow Z) \rightarrow (X \rightarrow Z) \wedge (Y \rightarrow Z) \\
\text{by} \rightarrow \textbf{-I}-\text{tac}_2 \text{ at } \mathrm{U}_1 \\
\quad x \in (X+Y) \rightarrow Z \gg (X \rightarrow Z) \wedge (Y \rightarrow Z) \\
\quad \text{by } \wedge\textbf{-I}-\text{tac}_2 \\
\quad\quad x \in (X+Y) \rightarrow Z \gg X \rightarrow Z \\
\quad\quad \text{by} \rightarrow \textbf{-I}-\text{tac}_2 \text{ at } \mathrm{U}_1 \\
\quad\quad\quad x \in (X+Y) \rightarrow Z, y \in X \gg Z \\
\quad\quad\quad \text{by} \rightarrow \textbf{-E}_2-\text{tac}_2 \text{ on } x \\
\quad\quad\quad\quad x \in (X+Y) \rightarrow Z, y \in X \gg X+Y \\
\quad\quad\quad\quad \text{by } +\textbf{-I}_1-\text{tac}_2 \text{ at } \mathrm{U}_1 \\
\quad\quad\quad\quad\quad x \in (X+Y) \rightarrow Z, y \in X \gg X \\
\quad\quad\quad\quad\quad \text{by Hyp}-\text{tac}_2 \\
\quad\quad\quad\quad\quad x \in (X+Y) \rightarrow Z, y \in X \vdash X \in \mathrm{U}_1 \\
\quad\quad\quad\quad\quad\quad r_1 \\
\quad\quad\quad\quad x \in (X+Y) \rightarrow Z, y \in X, z \in Z \gg Z \\
\quad\quad\quad\quad \text{by Hyp}-\text{tac}_2 \\
\quad\quad\quad x \in (X+Y) \rightarrow Z \vdash X \in \mathrm{U}_1 \\
\quad\quad\quad\quad r_2 \\
\quad\quad x \in (X+Y) \rightarrow Z \gg Y \rightarrow Z \\
\quad\quad \text{by} \rightarrow \textbf{-I}-\text{tac}_2 \text{ at } \mathrm{U}_1 \\
\quad\quad\quad x \in (X+Y) \rightarrow Z, y \in Y \gg Z \\
\quad\quad\quad \text{by} \rightarrow \textbf{-E}_2-\text{tac}_2 \text{ on } x \\
\quad\quad\quad\quad x \in (X+Y) \rightarrow Z, y \in Y \gg X+Y \\
\quad\quad\quad\quad \text{by } +\textbf{-I}_2-\text{tac}_2 \text{ at } \mathrm{U}_1 \\
\quad\quad\quad\quad\quad x \in (X+Y) \rightarrow Z, y \in Y \gg Y \\
\quad\quad\quad\quad\quad \text{by Hyp}-\text{tac}_2 \\
\quad\quad\quad\quad\quad x \in (X+Y) \rightarrow Z, y \in Y \vdash Y \in \mathrm{U}_1 \\
\quad\quad\quad\quad\quad\quad r_3 \\
\quad\quad\quad\quad x \in (X+Y) \rightarrow Z, y \in Y, z \in Z \gg Z \\
\quad\quad\quad\quad \text{by Hyp}-\text{tac}_2 \\
\quad\quad\quad x \in (X+Y) \rightarrow Z \vdash Y \in \mathrm{U}_1 \\
\quad\quad\quad\quad r_4 \\
\quad \vdash X \in \mathrm{U}_1 \\
\quad\quad r_5
\end{array}
$$

Figure 7.11: A tactic tree

$\Gamma \gg \mathrm{I}(\lambda x.b, A \rightarrow B)$	$[\rightarrow\textbf{-I}^1(\Gamma \vdash \mathrm{r} \in \mathrm{I}(\lambda x.b, A \rightarrow B), d_1, d_2)]$
by $\rightarrow\textbf{-I}^1$–tac with U_i	
$\quad\Gamma, x \in A \gg \mathrm{I}(b, B)$	$[d_1 \text{ for r}]$
$\quad\Gamma \gg \mathrm{I}(A, \mathrm{U}_i)$	$[d_2 \text{ for r}]$
$\Gamma \gg A \rightarrow B$	$[\rightarrow\textbf{-I}^2(\Gamma \vdash \lambda x.b \in A \rightarrow B, d_1, d_2)]$
by $\rightarrow\textbf{-I}^2$–tac with U_i	
$\quad\Gamma, x \in A \gg B$	$[d_1 \text{ for } b]$
$\quad\Gamma \gg \mathrm{I}(A, \mathrm{U}_i)$	$[d_2 \text{ for r}]$
$\Gamma \gg \mathrm{I}(\langle a, b\rangle, A \wedge B)$	$[\wedge\textbf{-I}^1(\Gamma \vdash \mathrm{I}(\langle a, b\rangle, A \wedge B), d_1, d_2)]$
by $\wedge\textbf{-I}$–tac^1	
$\quad\Gamma \gg \mathrm{I}(a, A)$	$[d_1 \text{ for r}]$
$\quad\Gamma \gg \mathrm{I}(b, B)$	$[d_2 \text{ for r}]$
$\Gamma \gg A \wedge B$	$[\wedge\textbf{-I}^2(\Gamma \vdash \langle a, b\rangle \in A \wedge B, d_1, d_2)]$
by $\wedge\textbf{-I}^2$–tac	
$\quad\Gamma \gg A$	$[d_1 \text{ for } a]$
$\quad\Gamma \gg B$	$[d_2 \text{ for } b]$

Figure 7.12: A few $\mathcal{B}_3^{\mathbf{TT}}$ tactics.

This first issue can be addressed by using the **I-I** and **I-E** rules which allow us to reflect the judgment form $\Gamma \vdash a = b \in A$ into the logic as $\Gamma \vdash \mathrm{r} \in I(a, b, A)$. For example, the following rules can be derived

$$\rightarrow\text{-}\mathbf{I}^1 \quad \frac{\Gamma \vdash \mathrm{r} \in \mathrm{I}(A, \mathrm{U}_i) \quad \Gamma, x \in A \vdash \mathrm{r} \in \mathrm{I}(b, B)}{\Gamma \vdash \mathrm{r} \in \mathrm{I}(\lambda x.b, A \rightarrow B)}$$

$$\rightarrow\text{-}\mathbf{I}^2 \quad \frac{\Gamma \vdash \mathrm{r} \in \mathrm{I}(A, \mathrm{U}_i) \quad \Gamma, x \in A \vdash b \in B}{\Gamma \vdash \lambda x.b \in A \rightarrow B}$$

$$\wedge\text{-}\mathbf{I}^1 \quad \frac{\Gamma \vdash \mathrm{r} \in \mathrm{I}(a, A) \quad \Gamma \vdash \mathrm{r} \in \mathrm{I}(b, B)}{\Gamma \vdash \mathrm{r} \in \mathrm{I}(\langle a, b\rangle, A \times B)}$$

$$\wedge\text{-}\mathbf{I}^2 \quad \frac{\Gamma \vdash a \in A \quad \Gamma \vdash b \in B}{\Gamma \vdash \langle a, b\rangle \in A \times B}$$

from which the tactics of Figure 7.12 can be defined.

The second issue mentioned above can be addressed in a manner similar to the approach taken in the forth encoding of Hoare logic. That is, by defining an encoding of **TT** with an event set that does not contain derivations.

7.8 Nuprl refinement revisited

The previous section demonstrated that refinement in Nuprl could be modeled with tactic trees constructed over a collection of tactics derived from a formal system **TT**. This collection of tactics was designed to provide for the extraction of terms through the composition of validations as well as a clear proof-theoretic picture of the logic represented in the refinement system. A tactic-tree based implementation of this type theory would differ in several respects from the apparatus of refinement currently implemented in the Nuprl system. This section will explore these differences and discuss the trade-offs involved in the tactic-tree approach.

First, a closer look at the implementation of refinement in Nuprl is required. An idealized description is presented that is faithful to Nuprl while ignoring many unessential details.

The types of goal, event, tactic, and refinement trees (called proofs) are defined in a mutually dependent fashion in the current implementation of Nuprl. In Nuprl the type `tactic` is defined as

```
tactic = proof -> proof list # validation
```

where

```
validation = proof list -> proof
```

and where proof is the type of (possibly incomplete) refinement trees. Goals in Nuprl are not sequents but refinement trees. Events in Nuprl are also refinement trees. A refinement tree p achieves a p' if p is complete and both trees have the same sequent at the root. Note that there is no validation in a Nuprl refinement tree since a complete refinement tree containing only basic rules achieves itself. An object of type proof is either of the form

$$\text{prl-leaf}(g)$$

or a node of the form

$$\text{prl-node}(g, R, (p_1, \ldots, p_n))$$

where g is a sequent, the p_i are proofs and R is a *rule*. Using our notation from the previous chapter, a tactic T with

$$T(g) = ((g_1, \ldots, g_n), v)$$

would be represented in Nuprl by a tactic T'

$$T'(\text{prl-leaf}(g)) = ((\text{prl-leaf}(g_1), \ldots, \text{prl-leaf}(g_n)), v')$$

that takes the refinement tree $p = \text{prl-leaf}(g)$ as input and returns as subgoals a list of degenerate refinement trees. However, in general a tactic can analyze an entire refinement tree

$$T'(p) = ((\text{prl-leaf}(g_1), \ldots, \text{prl-leaf}(g_n)), v').$$

In both cases the system rejects applications that do not conform to

$$\text{fringe}(v'(\text{prl-leaf}(g_1), \ldots, \text{prl-leaf}(g_n))) = (g_1, \ldots, g_n).$$

and

$$\text{root}(v'(\text{prl-leaf}(g_1), \ldots, \text{prl-leaf}(g_n))) = \text{root}(p).$$

A rule is either a basic rule of the form rule$(rn, a_1, ...a_n)$ where rn is a rule name and the a_i are arguments, or a tactic call of the form tactic(T, t) where t is a refinement tree.

The Nuprl system supports two modes of using a given tactic T'. In one mode of use T' is viewed as a *refinement tactic*. If the user has selected a (possibly degenerate) proof tree p and invokes the tactic T' the following actions occur. First, the tactic T' is applied to prl-leaf(g) where g is the root sequent of p

$$T'(\text{prl-leaf}(g)) = ((\text{prl-leaf}(g_1), \ldots, \text{prl-leaf}(g_n)), v')$$

returning a list of degenerate proof trees and a validation function mapping proof trees to a proof tree. The following node is then inserted into the refinement tree replacing p

$$\text{prl-node}(g, \text{tactic}(T', t), (\text{prl-leaf}(g_1), \ldots, \text{prl-leaf}(g_n)))$$

where

$$t = v'(\text{prl-leaf}(g_1), \ldots, \text{prl-leaf}(g_n)).$$

(Actually, this node is inserted only if the root of t is g, otherwise p remains unchanged.)

In the second mode of use, the tactic T' can be used as a *transformation tactic.* If the user has selected a proof tree p and invokes the tactic T' the following actions occur. First, the tactic T' is applied to p

$$T'(p) = ((\text{prl-leaf}(g_1), \ldots, \text{prl-leaf}(g_n)), v')$$

returning a list of degenerate refinement trees and a validation function mapping proof trees to a proof tree. The subtree p is then replaced with the proof tree

$$v'(\text{prl-leaf}(g_1), \ldots, \text{prl-leaf}(g_n))$$

if its root is g. Note that a transformation tactic has access to the entire proof tree p.

This arrangement has the advantage of allowing for a uniform treatment of tactics that are used as rules and tactics that are used as proof tree transformations. This would be lost in a reimplementing Nuprl in the tactic tree framework where such functions would be distinguished by type as well as by use.

7.9 Concluding remarks

In the tactic-tree framework, refinement is viewed as a deductive apparatus arising from the use of tactics to construct tactic trees rather than as an extension to the deductive apparatus of a *refinement logic.*

There are several advantages to this point of view. First, it provides a logic-independent explanation of refinement which is especially advantageous with respect to implementation. The representation of tactic trees and a structure-oriented interface for them can be treated independently, without regard to a particular object logic, and implemented as a generic reusable component. This done, one needs only to define a set of basic tactics to obtain a *refinement editor.*

Second, it provides a framework in which to explore the relationship between a logic and its various refinement presentations. Each such presentation originates from a set of formal derivations and is induced by a different set of design choices. This places emphasis on the proof-theoretic underpinnings of this design activity and brings to the fore such issues as the soundness and completeness of a given presentation. It allows for a clear and uniform explanation of the relationship between a logic and its refinement rules, and issues such as extraction. For example, in Nuprl a separate mechanism is required to extract terms from refinement proofs. In the tactic tree approach this is accomplished in a more uniform way with the implementation of syn(r) — essentially the same method of extracting programs used in our encoding of Hoare logic.

Chapter 8

Directions for future research

The problems of providing logic-independent accounts for notational definition and top-down refinement have been addressed. The solutions presented in the preceding chapters are intended to aid in the design and implementation of interactive proof development systems. This chapter explores directions for future research, first with regard to notational definition, and then with regard to abstract refinement.

8.1 Notational definition

The formal account of notational definition was presented at a fairly high level and ignored many of the issues involved in actually implementing mechanisms to support notational definition. Much work remains to be done in this area.

The Nuprl-style of declaration outlined in Section 5.1 has the advantage of combining the declaration of new notation with the declaration of the display of this notation. This works well for noniterative notations but it is not clear how to extend it in a general way to iterative notations (Section 3.6). Another interesting problem involves the design of a declaration facility that takes advantage of recent advances in high-resolution bitmap displays so that two dimensional notations could be defined.

The acceptable expansion Exp^{Δ} defined in Chapter 4 has been used as the basis for the expansion procedure of the EFS (see Section 5.4 and [Gri87]). However, this function is not *optimal* in the sense that it may α-convert a variable on expansion that would not have clashed with other variables. So a question naturally arises: can an acceptable expansion be defined that is optimal in this sense?

The functions of substitution, α-conversion, and the determination of free variables can be applied directly to instances of defined notations. In an implementation of a language, there may be other functions defined on expressions that could be extended in such a way that they are defined directly on instances of defined notations; that is, that the application of such a function would not require expansion of definition instances. For example, in the implementation of the EFS there is a type inference function (that is, type inference at the level of the $\lambda\Pi$ language, and not at the level of its syntactic encoding) defined on basic expressions. In the EFS this functions has been defined to expand definitions only when required, although in general this can involve expanding all instances of defined notation in an expression. One possible solution is to attach some information, generated at declaration time, to a defined notation that and could be used by a type inference procedure to compute the type of an instance of this notation without expansion. It is not clear how to address this kind of problem in general. Perhaps it is it possible to identify a class of functions that can be automatically extended to defined notations in some way.

One possible application of the formal account, outline in Chapters 3 and 4, is in the design of well-behaved macro facilities for certain programming languages. One obvious candidate is λ-Prolog [MN86,Nad87], which is a higher-order Prolog based on the simply typed λ-calculus. This would require some work to ascertain the best way in which to modify higher-order unification in order to accommodate instances of defined notations.

Application to macro facilities for general programming languages seems more difficult. For example, Lisp macros are not abbreviative in nature but are essentially very general program-valued programs. A preprocessor executes these macro programs to eliminate all macros before compilation. Macro elimination may involve the unintended capture of free variables of arguments to a macro, and so poses a problem somewhat similar to one of the problems addressed above for notational definition. This problem has been addressed by [KFFD87] with their definition of "hygienic macro expansion" for Lisp.

8.2 Abstract refinement

One intriguing possible extension involves the use of the structure-oriented tactic tree (refinement) editor for the *definition* of tactics. This is suggested by the definition in Section 6.2 of the function $\mathrm{tsyn}(r)$ that extracts a tactic from a tactic tree r. Recall that if r is a tactic tree and $T = \mathrm{tsyn}(r)$, then

$$T(\mathrm{root}(r)) \quad = \quad ((g_1, \ldots, g_n), v),$$

where $(g_1, \ldots, g_n) = \mathrm{fringe}(r)$. This suggests the possibility of defining the tactic T simply by using a structure-oriented tactic tree editor to construct r. For example, suppose, given the description of type theory in Section 7.7 and the new notation $A \leftrightarrow B$ abbreviating $A \to B \wedge B \to A$, that we wanted to define the tactic `iff-intro-tac` as

$\Gamma \gg A \leftrightarrow B$
by `iff-intro-tac`
  $\Gamma, x \in A \gg B$
  $\Gamma, y \in B \gg A$

We could, as is currently the practice, write this tactic in ML as

```
\g. if is_def ``iff'' (concl g)
    then (def_elim ``iff''
          THEN (Prod_Intro_2
                THENL [ (Arrow_Intro_2 THEN ITAC);
                        (Arrow_Intro_2 THEN ITAC)])
          ) g
    else fail
```

where `is_def`, `def_elim`, and `concl` are assumed to be defined in the obvious way. But this is, more or less, the tactic that is extracted from the incomplete tactic tree

$\Gamma \gg A \leftrightarrow B$
by def-elim with iff
  $\Gamma \gg A \rightarrow B \wedge B \rightarrow A$
  by $\wedge$-**I**—tac$_2$
    $\Gamma \gg A \rightarrow B$
    by $\rightarrow$ -**I**—tac$_2$
      $\Gamma, x \in A \gg B$
      by ?
    $\Gamma \gg B \rightarrow A$
    by $\rightarrow$ -**I**—tac$_2$
      $\Gamma, y \in B \gg A$
      by ?

That is, the tactic `iff-intro-tac` could be *extracted* from this tactic tree. However, this example is only meant to be suggestive. In general, the situation is more complicated since we would want to define parameterized tactics and use parameterized tactics in the construction of r. It is not clear how to address these complications in a general way.

On a related topic, the notation defined in Section 7.2 and used in Sections 7.6 and 7.7 for the definition of basic tactics could be integrated into the logic specification interface of the tactic tree implementation described above. That is, a logic implementor could use a structure-oriented editor to enter the notation

$$\begin{array}{ll}
\Gamma \gg A \wedge B & [(\wedge\text{-}\mathbf{I}(\Gamma \vdash \langle a, b\rangle \in A \wedge B, d_1, d_2), \langle a, b\rangle)] \\
\text{by } \wedge\text{-}\mathbf{I}\text{—tac}_2 & \\
\quad \Gamma \gg A & [(d_1, a)] \\
\quad \Gamma \gg B & [(d_2, b)]
\end{array}$$

rather than defining the new tactic in ML or some other implementation language. This notation could then be translated into a basic tactic in the implementation language.

A proof development system may support several different mechanisms for proof construction. The fact that the notion of refinement has been decoupled from the proof system suggests the possibility of encorporating such mechanisms directly into an implementation of tactic trees. That is, subtrees that are not themselves tactic trees could be attached to tactic trees. These subtrees could then construct events achieving goals in some other way. One example would be a mechanism that allows a user to directly construct events (say, "bottom-up" derivations in some formal system). Another possibility is to extend tactic trees with subtrees that are generated by an attribute grammar. Reps and Alpern [RA84] introduced a scheme for representing goal-directed proofs as an attribute grammar having one grammar production for each refinement rule and having the relationship between goals and subgoals defined by attribute equations. Proof editors implemented using this scheme allow for a flexible style of interactive proof construction. Goal-directed proof trees can pass through inconsistent states and contain incomplete terms. For example, a proof can proceed even when the initial goal is incomplete. Later changes to the goal are then propagated through the tree. The Reps-Alpern approach was extended by the author and used with good results in the EFS. However, it is far from clear whether this can be integrated smoothly into a tactic-based system.

The account of notational definition and abstract refinement has been referred to as *logic-independent.* However, these frameworks are not only logic-independent but *independent of logic.* While the motivation for developing these frameworks has been the design and implementation of interactive proof development systems, this does not limit the scope of their potential applications.

Bibliography

[AHM87] Arnon Avron, Furio Honsell, and Ian Mason. Using typed lambda calculus to implement formal systems on a machine. LFCS Report Series ECS-LFCS-87-31, Laboratory for the Foundations of Computer Science, Edinburgh University, 1987.

[All87a] Stuart Allen. A non-type-theoretic definition of martin-löf's types. In *Proceeedings of the Second Symposiuym on Logic in Computer Science*, pages 215–224. IEEE, 1987. Cornell TR 87-832.

[All87b] Stuart Allen. *A Non-Type-Theoretic Semantics for Type-theoretic Language.* Ph.D. dissertation, Cornell University, Ithaca, NY, 1987. Cornell TR 87-866.

[And86] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, 1986.

[Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: A survey – Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, 1981.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, Amsterdam, revised edition, 1984.

[Bas88] David Basin. An environment for automated reasoning about partial functions. In *CADE-9*, 1988. Cornell TR 87-884.

[Bat79] Joseph L. Bates. *A Logic For Correct Program Development.* Ph.D. dissertation, Cornell University, 1979.

[BC83] Joseph L. Bates and Robert L. Constable. The nearly ultimate PRL. Technical Report TR-83-551, Cornell University, Department of Computer Science, Cornell University, Ithaca, NY, 1983.

[BC85] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Trans. Prog. Lang and Sys.*, 7(1):113–136, 1985.

[Bur86] R.M. Burstall. Research in interactive theorem proving at Edinburgh University. LFCS Report Series ECS-LFCS-86-12, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1986.

[CF58] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume I. North-Holland, Amsterdam, 1958.

[CH85] Thierry Coquand and Gérard Huet. Constuctions: A higher order proof system for mechanizing mathematics. In Bruno Buchberger, editor, *EUROCAL '85*, volume 203 of *Lecture Notes in Computer Science*, pages 151–184. Springer-Verlag, 1985.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CKB85] Robert L. Constable, Todd B. Knoblock, and Joseph L. Bates. Writing programs that construct proofs. *Journal of Automated Reasoning*, 1:285–326, 1985.

[Cle87] Rance C. Cleaveland. *Type-Theoretic Models of Concurrency.* Ph.D. dissertation, Cornell University, 1987.

[Con86] Robert L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[Coq85] Thierry Coquand. Une théorie des constructions. Phd thesis, Université Paris, 1985.

[dB70] N. G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, editors, *Symposium on Automated Demonstration*, volume 125 of *Lecture Notes in Mathematics*, Berlin, December 1970. Springer-Verlag.

[dB80] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[DRT81] A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Conference Record of the Eigth Annual ACM Symposium on Principles of Programming Languages*, pages 105–116, 1981.

[Dum77] Michael Dummett. *Elements of Intuitionism.* Oxford University Press, 1977.

[dV87] R.C. de Vrijer. *Surjective Pairing and Strong Normalization: Two Themes in the Lambda Calculus.* University of Amsterdam, 1987. PhD Thesis.

[GMW79] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF : A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.

[Gri87] Timothy G. Griffin. An environment for formal systems. Technical Report 87-846, Department of Computer Science, Cornell University, 1987. (also LFCS report ECS-LFCS-87-34, Department of Computer Science, University of Edinburgh).

[Gri88a] Timothy G. Griffin. A formal account of notational definition. Technical Report 88-901, Department of Computer Science, Cornell University, 1988.

[Gri88b] Timothy G. Griffin. Notational definitions — a formal account. In *Proceedings of the Third Symposium on Logic in Computer Science*, July 1988.

[Har85] Robert W. Harper. *Aspects of the Implementation of Type Theory.* Ph.D. dissertation, Cornell University, 1985.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings of the Second Symposium on Logic in Computer Science*, 1987.

[HL78] Gérard Huet and Bernard Lang. Proving applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[HMM86] Robert W. Harper, David MacQueen, and Robin Milner. Standard ML. LFCS Report Series ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1986.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 1969.

[How86] Douglas J. Howe. Implementing number theory, an experiment with nuprl. In *Proceedings of the Eighth International Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 404–415. Springer-Verlag, 1986.

[How87] Douglas J. Howe. The computational behaviour of Girard's paradox. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 205–214. IEEE, 1987.

[How88] Douglas J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory.* Ph.D. dissertation, Cornell University, 1988.

[HS86] R. J. Hindley and J.P Seldin. *Introduction to Combinators and λ-Calculus.* London Mathematical Society Student Texts. Cambridge University Press, 1986.

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical computer science*, 1:25–57, 1975.

[Jon87] Claire Jones. The interactive proof editor. Technical Report LFCS-TN-11, Department of Computer Science, University of Edinburgh, 1987.

[KFFD87] Eugene Kohlbecker, Daniel Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, 1987.

[Kre86] Christoph Kreitz. Constructive automata theory implemented with the Nuprl proof development system. Technical Report TR 86–779, Department of Computer Science, Cornell University, 1986.

[LPR+88] Peter Lee, Frank Pfenning, John Reynolds, Gene Rollins, and Dana Scott. Research on semantically based program-design environments : The ergo project in 1988. Technical Report CMU-CS-88-118, Department of Computer Science, Carnegie Mellon University, 1988.

[Mil87] Robin Milner. Dialogue with a proof system. In H. Ehrig, R. Kowalski, G. Levi, and U. Montarari, editors, *TAPSOFT '87*, volume 249 of *Lecture Notes in Computer Science*, pages 271–275, Berlin, 1987. Springer-Verlag.

[ML] Per Martin-Löf. Intuitionistic type theory. Notes by Giovanni Sambin of a series of lectures given in Padova, June 1980. Unpublished manuscript.

[ML71] Per Martin-Löf. A theory of types. Report 71–3, Department of Mathematics, University of Stockholm, February 1971.

[ML73] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H. F. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118, Amsterdam, 1973. North-Holland.

[ML82] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[MN86] Dale A. Miller and Gopalan Nadathur. Higher-order logic programming. In *Proceedings of the Third International Conference on Logic Programming.* Springer-Verlag, July 1986.

[Nad87] Gopalan Nadathur. *A Higher-Order Logic as the Basis for Logic Programming.* Ph.D. dissertation, University of Pennsylvania, 1987.

[NPS86] Bengt Nordström, Kent Petersson, and Jan Smith. *An introduction to Martin-Löf's Type Theory (summmer 1986 draft).* Department of Computer Sciences, University of Göteborg, Chalmers University of Technology, S-412 96 Göteborg, Sweden, 1986. in preperation.

[Pau85] L.C. Paulson. Interactive theorem proving with cambridge LCF : A user's manual. Technical Report Report 80, Computer Lab., University of Cambridge, 1985.

[Pau86] L.C. Paulson. Natural deduction proof as higher-order resolution. *Journal of Logic Programming*, 3(3):237–258, 1986.

[PE88] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. Technical Report ERGO-88-036, Department of Computer Science, Carnegie Mellon University, 1988.

[Pet82] Kent Petersson. A programming system for type theory. Technical Report 21, Programming Methodology Group, University of Göteborg/Chalmers Institute of Technology, 1982.

[Pot78] Garrel Pottinger. Proofs of the normalization and church-rosser theorems for a typed λ-calculus. *Notre Dame Journal of Formal Logic*, 19(3):445–451, 1978.

[Pot81] Garrel Pottinger. The church-rosser theorem for the typed λ-calculus with surjective pairing. *Notre Dame Journal of Formal Logic*, 22(3):264–268, 1981.

[Qui53] Willard Quine. *From a Logical point of view.* Harper and Row, 1953.

[RA84] Thomas W. Reps and Bowen Alpern. Interactive proof checking. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, 1984.

[Rep85] Thomas W. Reps. *Generating Language-Based Environments.* The MIT Press, 1985.

[Rey81] J. C. Reynolds. The essence of algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, 1981.

[RT84] Thomas W. Reps and T. Teitelbaum. The synthesizer generator. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Envirionments, Pittsburgh, Penn., April 23–25, 1984*, pages 42–48, May 1984. ACM SIGPLAN Notices 19, 5.

[RT85] Thomas W. Reps and T. Teitelbaum. The synthesizer generator reference manual. Dept. of Computer Science, Cornell University, Ithaca, NY,14853, 1985. Second Edition, 1987.

[RT88a] T. Reps and Teitelbaum T. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Springer-Verlag, 1988.

[RT88b] T. Reps and Teitelbaum T. *The Synthesizer Generator Reference Manual.* Springer-Verlag, 1988. Third Edition.

[Sta81] R.M. Stallman. EMACS, the extensible, customizable self-documenting display editor. In *Proceedings of the ACM SIGPLAN-SIGOA Symposium on Text Manipulation, SIGPLAN Notices*, volume 16, pages 147–156, 1981.

[Tro86] A. S. Troelstra. Strong normalization for typed terms with surjective pairing. *Notre Dame Journal of Logic*, 27(4):547–550, 1986.

[vD80] Diedrik van Daalen. The language theory of automath. Technical report, Technical University of Eindhoven, Eindhoven, The Netherlands, 1980. PhD thesis.

[WR27] Alfred N. Whitehead and Bertrand Russell. *Principia Mathematica*, volume 1. Cambridge University Press, 2 edition, 1927.