IMPLEMENTING DIRECT REFLECTION IN NUPRL

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

Eli Barzilay

January 2006

© 2006 Eli Barzilay
ALL RIGHTS RESERVED

IMPLEMENTING DIRECT REFLECTION IN NUPRL

Eli Barzilay, Ph.D.
Cornell University 2006

Reflection is the ability of some entity to describe itself. In a logical context, it is the ability of a logic to reason about itself. Reflection is, therefore, placed at the core of meta-mathematics, making it an important part of formal reasoning; where it revolves mainly around syntax and semantics — the main challenge is in making the syntax of the logic become part of its semantic domain.

Given its importance, it is surprising that logical computer systems tend to avoid the subject, or provide poor tools for reflective work. This is in sharp contrast to the area of programming languages, where reflection is well researched and used in a variety of ways where it plays an central role. One factor in making reflection inaccessible in logical systems is the relative difficulty that is immediately encountered when formalizing syntax: dealing with formal syntax means dealing with structures that involve bindings, and in a logical context it seems natural to use the same formal tools to describe syntax — often limiting the usability of such formalizations to specific theories and toy examples. Gödel numbers are an example for a reflective formalism that serves its purpose, yet is impractical as a basis for syntactical reasoning in applied systems.

In programming languages, there is a simple yet elegant strategy for implementing reflection: instead of making a system that describes itself, the system is made available to itself. We name this *direct reflection*, where the representation of language features via its semantics is actually part of the semantics itself — unlike the usual practice in formal systems of employing indirect reflection. The advantages of this approach is the fact that both the system and its reflected counterpart are inherently identical, making for a lightweight implementation.

In this work we develop the formal background and the practical capabilities of an applied system, namely Nuprl, that are needed to support direct reflection of its own syntax. Achieving this is a major milestone on the road for a fully reflected logical system. As we shall demonstrate, our results enable dealing with syntactical meta-mathematical content.

BIOGRAPHICAL SKETCH

Eli Barzilay was born in Israel in the summer of 1970. In 1980 a coincidence made him switch from Astrophysics to Computer Science as his destination. In 1991 he began his academic education in the Mathematics department of Ben-Gurion University, earning a Bachelor of Science in 1994 and a Master of Science in 1997. At that same year he moved to Ithaca, NY, and began a long chapter of his life that ends in this text. In March 2005 he sat down in front of a blank Emacs screen and typed this paragraph.

For my father,
and for Tali.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Robert Constable, for being able to work on an exciting topic; members on my committee, Greg Morrisett, who provided many enjoyable hours of PL flames, and Barry Perlus for opening my eyes in the digital sense; Stuart Allen, who spent months and years working with me; Aleksey Nogin, who provided motivation in both early and late stages of my work. I also thank the Nuprl group as a whole, who provided extensive feedback during many PRL seminars, and off-line discussions. This work was supported by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the Office of Naval Research under Grant #N00014-01-1-0765, and NSF Innovative Programming technology for embedded systems #CCR-0208536.

But most of all I thank my wife, Regina, who deserves more than these words.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1
Introduction

Reflection is the ability of some entity to describe itself. It is a deep idea in logic, computer science, linguistics, philosophy, art, and more. Reflection has been a source of philosophical discussions for ages, with the liar paradox and variations (*e.g.*, "This sentence is false.") being the best known example — allegedly originated by the Greek philosopher Eubulides the Megarian in the fourth century B.C.[1], studied in the twelfth century under the name of "insolubles" or "insolubilia" [68], etc. Reflection is also a very current topic in programming languages — from using macros to enhance expressiveness [30], to using theorem provers to reason about programming languages, recently becoming a renewed focus of attention due to the PoplMark challenge [7].

In a logical context, reflection is the ability of a logic to reason about itself. Reflection is, therefore, placed at the core of meta-mathematics, making it an important part of formal reasoning; where it revolves mainly around syntax and semantics — the main challenge is in making the syntax of the logic become part of its semantic domain. For theorem provers, reflection is not only a way to formalize meta-mathematical content, it can be used to shorten proofs, extend the system with verified tactics, provide meta-proof tools, and reason about complexity [6, 47], as well as promoting reasoning about syntax which is necessary for formal programming languages research. Indeed, within the Nuprl group at Cornell there have been attempts at implementing reflection since the 1980s.

Given its importance, it is surprising that program verification systems [35] tend to avoid the subject, or provide poor tools for reflective work. This is in sharp contrast to the area of programming languages, where reflection is well researched and used in a variety of ways where it plays an central role.

Verification systems are inherently close to programming languages: relationships between the concepts of proof systems and programming languages are well studied. It is therefore quite common to find ideas from one field inspiring advancements in the other. One factor in making reflection inaccessible in logical systems is the relative difficulty that is immediately encountered when formalizing syntax: dealing with formal syntax means dealing with structures that involve bindings, and in a logical context it seems natural to use the same formal tools to describe syntax — often limiting the usability of such formalizations to specific theories and toy examples. Gödel numbers are an example for a reflective formalism that serves its purpose, yet is impractical as a basis for syntactical reasoning in applied systems.

In programming languages, there is a simple yet elegant strategy for implementing reflection: instead of making a system that describes itself, the system is made available to itself. We name this *direct reflection*, where the representation

[1] It seems that the paradox by the Cretan philosopher Epimenides of Knossos predates Eubulides, but was not originally intended as a paradox.

of language features via its semantics is actually part of the semantics itself — unlike the usual practice in formal systems of employing indirect reflection such as Gödel numbers. The advantages of this approach is the fact that both the system and its reflected counterpart are inherently identical, making for a lightweight implementation.

In this work we will thoroughly discuss reflection approaches, focusing on direct reflection as the superior solution for implementing reflection. This continues the tradition of extending the synergy between the two fields: we will aim at achieving a reflection strategy that is as elegant and as robust as the common approach in programming languages. Based on this discussion, we will formalize and implement a syntactic reflection mechanism for a proof development system, namely Nuprl, and demonstrate the practicality of our approach. The discussion will revolve around Nuprl as a known representative of theorem proving systems, yet the general principles are applicable for similar applications.

The benefits of having a reflection mechanism was shown as an extremely useful tool in numerous domains, not only programming languages, but other substrate systems as well — operating systems, object systems, data bases etc. Theorem provers in general, and Nuprl in specific, are substrate systems of yet another kind, and as such, it will benefit as well from a reflection mechanism. In these systems, reflection can, again, have several different meanings, from reflecting facts about the language and the computations that make the system to reflecting the logic itself, which makes it possible to perform meta-reasoning. The global goal of our work will be providing a method for implementing syntactic reflection in logical system — given that any form of logical reflection is preconditioned by the ability to reflect syntax, our work will form a basis for future reflective extensions of any kind. More specifically, this work is part of an overall effort to get a practical reflection of syntax, computation and proof in Nuprl [16, 6]. Reflecting syntax in a logical system entails writing proof rules that express that reflection, *i.e.*, establishing an inferential connection between the actual syntax used and the meta-terms supposedly referring to it.

As we strive to import ideas from the world of programming language practice, the discussion will be kept close the implementation level. Reflection combines the implementation view with the theoretical view — where programming languages lean to the former and logical systems to the latter. Breaking this pattern we will draw ideas from actual implementation strategies, and use code snippets as an initial discussion seed. This follows the basic intuition that such ideas are most useful when they are implemented, leading to an "implementation as understanding" principle which guides this work.

1.1 Reflection and Art

This work is written by a person who considers programming an art. Programmers who hold this opinion often design code according to aesthetic considerations,

where often there is some solution that *feels* like "The Right Thing" (sometimes abbreviated as "TRT"). These programmers will then find great joy in observing how such beautiful designs make for programs that almost write themselves.

The main approach that is used in this work originates in such a design. The research was carried on based on the intuition that it is the right thing, even at times where we thought that we will not be able to express certain meta-proofs (Section 6.1). Later on, it turned out that good design did pay off.

For programmers, computer scientist, and logicians, such designs are apparent not only consciously, but also on an aesthetic, almost sub-conscious level. Certain programs, algorithms, and proofs are viewed in a very similar fashion to good pieces of art. In an attempt to demonstrate this, I have taken a minor in Cornell's Art department, and used photography to visually demonstrate deep ideas from computer science and logic. These photographs are intended to impact the viewer on both levels of aesthetics that were mentioned: at conscious level where the puzzle element is obvious, and at a sub-conscious level where aesthetics dominate. Six of these pictures appears on page 6 as a sample, all are connected to various aspects of this work, *e.g.*, syntax vs. semantics, language, different kinds of reflection and self reference.

1.2 Accomplishments

This thesis will present a practical way to reflect syntax in the Nuprl theorem prover. Several important concepts will be demonstrated:

- *Direct reflection* will be shown as an approach that brings similar wins for reflecting theorem provers, as it does for programming languages.

- Using direct reflection is feasible with a logical system. As is the case with programming language implementations (*e.g.*, most Scheme implementations), this results in a light-weight reflection system which is easy to implement and extend. Furthermore, such an implementation benefits the reflected system in numerous ways, the major one being the fact that the object and the meta levels share the same functionality which leads to a robust environment where no additional effort is needed in proving the two as equivalent.

- Higher-order abstract syntax techniques form a plausible system for efficient and practical reflection of syntax. Specifically, proofs that are based on concrete-syntax can be formalized in such a system in a natural way: quoted variable names behave just like Scheme symbols. We do not even require sophisticated pattern-matching rewrite methods to deal with quoted syntax, although our approach does not conflict with such method should there be a need for them.

- Using shifted operator names is a viable solution to the problem of reflecting a uniform language with semantics that forbid quoted contexts. The result

is just as convenient to use as using a quote context in Scheme, and it makes it very convenient to mix quoted terms with unquoted descriptions. In addition, the resulting quoting scheme not only makes it easy to reflect the same syntactic objects (making it a direct reflection), but it is also naturally efficient — there is no need to take special care in avoiding an exponential blowup: shifted content has exactly the same size as unshifted content.

This is relevant in a wider scope than Nuprl: it can be applied to other contexts in which referential transparency is required.

- Our HOAS formalization does not depend on types as the tool that forbids exotic terms. Instead, we use a syntactic approach that is used to restrict the construction of syntax-denoting syntax. Translated to a programming language context, this method implies a syntactic check that makes sure that shifted operators are well behaved — that bound variables are used as inert values, only being passed around to 'template holds', not being used for their value. This method can be used in other contexts, providing a lighter alternative to common type-centric approaches, especially in the context of dynamically typed languages.

- Our implementation is mostly independent of Computational Type Theory (CTT), no semantical changes to Nuprl are required for our implementation. This includes the fact that the reflected syntax is automatically open-ended, as it inherits its features from the Nuprl implementation.

  This makes it an attractive strategy for adding reflection to existing theorem provers and logical framework. The MetaPRL system has been recently extended with reflective features, based on the research that was presented here. It should be noted that the approach taken by the MetaPRL implementation simplifies things a little by using a '`bterm`' as a basic constructor for *bound terms* — this is a natural choice as the MetaPRL implementation uses bound terms as the basic type.

- For users of the Nuprl theorem prover, the most important aspect is the user interface. Our implementation covers all important aspect of the Nuprl interface, including careful attention to display forms for quoted terms which use colors to indicate 'quotedness' level.

1.3 Outline

The general structure of this text roughly follows the research path that it describes. Chapter 2 begins with a general discussion of reflection and related concepts, and introduces the main guidelines that the work follows; it is an extended introduction combined with the philosophical discussion and core around which the implementation will be founded on. Appendix A describes some of the terms that are used in our discussion, and is most relevant with this chapter. Getting on to a

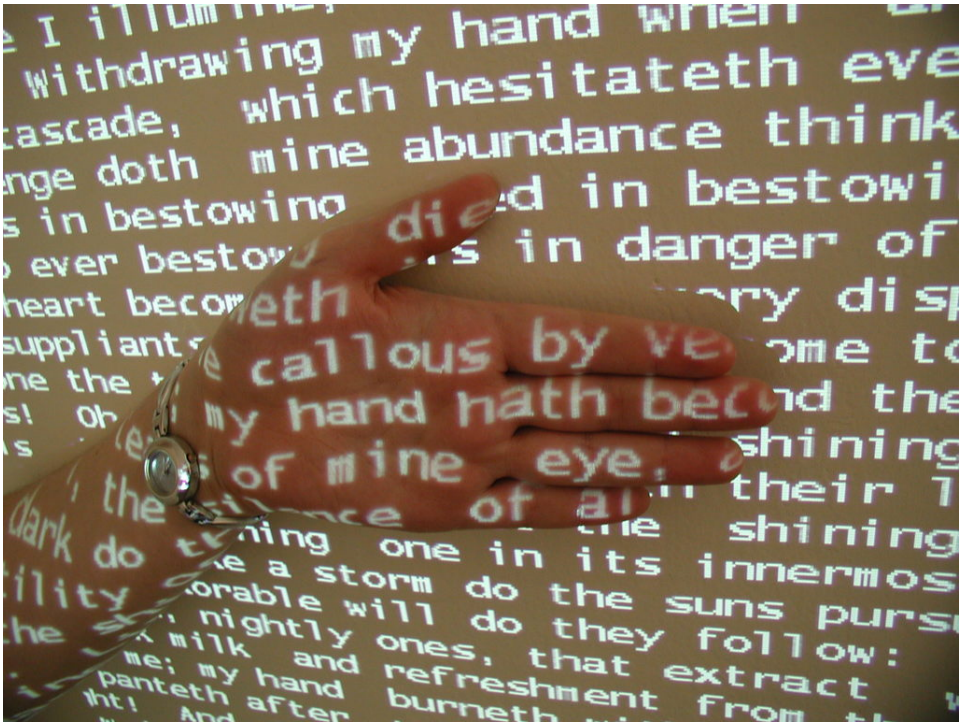
more concrete level, we consider various implementation strategies in Chapter 3, taking into account theoretical as well as practical implications of these choices. This leads to a natural choice for a directly reflected syntax scheme, which is thoroughly discussed, comparing its expressive power to known systems (mainly to programming languages that are directly reflective), and the system-level extension that implements this is described.

Having made the implementation choice, Chapter 4 continues by providing the basic semantical account of quoted syntax values; also mentioned in this chapter are a few common higher-level abstract syntax approaches, which are different from our account. In Chapter 5 the formal presentation is extended and additional technical detail are given. These two chapters are based on work with Stuart Allen.

Chapter 6 demonstrates a practical proof that can be accomplished with our new reflective extension — this chapter goes along two parallel levels: first, the practical proof is described and used as a motivation for additional functionality (encapsulated in a Nuprl theory), and on the second level, the issues that are related are examined and clarified. The relevant Nuprl material is included in two theory files, which are included for reference in Appendix B.

Finally, Chapter 7 concludes the thesis, highlighting the goals that were achieved, and summarizing additional required work and future research directions that have opened up as a result.


n'est pas une


Withdrawing my hand
which hesitateth
mine abundance
callous by
my hand hath
of mine eye,
shining
one in its
a storm do the suns
nightly ones, that extract
milk and refreshment
me; my hand
panteth after


(lambda (x) (x x))

Chapter 2
The Scope of Reflection

An implementation of a reflective system is in essence a merging of an *object* and a *meta language*, two different environments, each with its own notion of syntax and semantics. As a general phenomenon, reflection is any way which enables some entity to refer to itself. In the context of logic and computer science, this requires some form of meta-information *about* the entity in a form it can *use*. In any case, the meta information can be viewed as constituting a language, even when it is not obvious. For example, meta-data in a database is in the same form (language) as the database itself, and a reflective object system uses the language of objects to describe itself.

Syntax and semantics play a major role in this context: a syntactic entity refers to some ontological entity in its semantic domain through a semantics relation. The following discussion, therefore, starts with an examination of these attributes in a general linguistic context. We then move to formal languages — programming languages using a pure subset of Scheme as a concrete example, and logic, with Nuprl's Computational Type Theory (CTT) [18, 5] as our goal environment.

For the sake of clarity and completeness, the glossary in Appendix A should be consulted for meanings of terms (some have varying meanings in existing literature).

2.1 Reflection Applied to General Languages

The term "language" as we use it, is a *formal* way of communicating concepts — theoretical objects in some abstract domain. The language itself can come in several different ways such as vocal sounds, written text, or text encoded in computer files. Whatever form a language takes, there are rules to specify what constructs are correct — *syntax* rules, and how to associate syntactic constructs with the concepts they represent — *semantics* (or meaning). For example the syntactic construct of the Hebrew sound "shalosh", of the English letter sequence 't'-'h'-'r'-'e'-'e', of the ASCII character string "3" in some conventional programming language, and of the Nuprl term '`natnum{3:n}()`', all have the semantics of the number three. The semantic rules match syntactic structures in the language to objects in the domain that this language denotes.

Note that the place where syntax ends and semantics begins is not fixed — we decide what syntax is correct, and then how to get its semantics, so filtering out some constructs can be done by declaring them as syntactically incorrect *or* by making their semantics void. For example, we can either say that the expression '`1+"a"`' is syntactically incorrect, or that it is syntactically correct but raises an error when evaluated or compiled, making it meaningless. This is clear in a programming language implementation: deciding what component is responsible for detecting such errors — *e.g.*, the evaluator, the parser, or the type checker; but

it is also a question in natural languages: one option is that "books sky snail" is syntactically incorrect because it contains a sequence of three nouns, but another is that it is correct because all three words are spelled correctly; Chomsky's 1957 example, "Colorless green ideas sleep furiously", points at a related problem: there is no agreement whether this sentence has any semantics or not.

In our context of discussing languages, the term "reflection" stands for a system that must have at least the following two properties: it must allow syntax that denotes (by its semantics) its own syntactic constructs (quotations), and it must have some way of using these constructs. In written natural language, the first property is achieved using quotation symbols. These symbols specify that a piece of text is not to be taken as carrying meaning in the normal way, instead, it represents the actual language syntax. For example, the English word "water" stands for water, but the text "the English word 'water'" uses the actual word "water" as a piece of syntax[1]. It is obvious that quotations are a fundamental aspect of reflection. The second property is achieved by the fact that we can actually talk about these pieces of text as semantic entities. In the natural language context, this means having words such as "word", "sentence", and "meaning". The reader should note that this paragraph is itself a good demonstration of these two properties (and of the fact that more than two levels can be used).

Following the above, if we want a language that can reflect itself, the first thing we need is for its semantic domain to include representations of the syntactic objects of the language. In other words, make the set of syntax constructs a subset of the represented values domain. We also need some syntax for denoting these objects, call them quotations (in the natural language case these are quotation symbols). When we have such a piece of syntax S_1 that denotes (through its semantics) a data structure that represents a piece of syntax S_0, we say that S_1 is the *quotation* of S_0.

Many representations can be used to specify quotations. One obvious choice is taken from the informal usage of quotes and raw text in natural language: strings of symbols. However, this is an extremely poor representation for programming languages and logical systems since it does not reflect the inherently recursive nature of syntactical constructs[2]. A representation that is natural in the context of formal languages is using the language capabilities for compound data objects, *e.g.*, defining structure types or using tuples and lists, but this approach has problems too as we will see shortly. There are other options for quotations, like a quotation

[1]It is interesting to note that I have yet to find a dictionary that refers to itself in the entry for "dictionary". Moreover, on-line dictionaries are taken from pre-existing books, with a "dictionary" entry that specifies "a reference *book*".

[2]Natural language syntax is (implicitly) structured as well, but this structuring can be ambiguous which means that a recursive tree structure can be insufficient, but our goal is formal languages so we assume that (explicit) structure is desired, and avoid irrelevant natural language debates.

context, operator shifting[3], and preprocessing mechanisms. These options and more are discussed in Section 3.3.

In addition to syntax and semantics, most languages have an inherent evaluation process: we first receive the syntax, then understand what it denotes (if it makes sense) using the semantics of our language, and then we evaluate the result[4]. This is a mental process that starts with a sentence as a piece of syntax, converts it to a piece of semantic information, and then forms a final mental piece of information in our mind using some form of evaluation.

Evaluation can take several forms, for example — we can identify and expand definitions such as "Eli's wife" or pronouns like "you" and identify them with other concepts such as a known person named "Regina Barzilay". We can also use some logical rules that are part of our language like eliminating double negations. More rules that we use to build such a 'mental image' can come from the process in which this image is built, for example, adjectives specify object attributes, so they are order-independent (*e.g.*, "the big blue car" and "the blue big car" have the same meaning). Finally, some information is taken from rules of the physical world: we know that "mixing flour and eggs" is the same as "mixing eggs and flour", or that "a half-full glass" is the same as "a half-empty glass" (this, of course, can depend on the context in which it is used).

There are also rules that handle quotations: this is interesting since it is the way natural language implements linguistic self-reference. Quotations can be used as any other object, and they actually describe their contents: so the first thing that makes this similar to the world of programming is that (normal) evaluation does not occur inside quotes. As an example, the previous paragraph mentions several pieces of text that would evaluate to the same mental image *if* they were unquoted[5]. More rules involve referencing pieces of text, as in "The third word of this sentence", or direct evaluation using terms like 'meaning' as in: "The word 'word' stands for the concept of a word".

This leads us to a third property of a reflected language: when we have the above two — a syntax that represents syntax, and quotation rules — then it is possible to talk about the language within itself, but there is no real guarantee that the quoted language is *identical* to the language itself. Therefore, the third property is the *correspondence* between this representation and the language itself. This can be regarded as the guard-dog that makes sure represented objects behave as we *expect* them to behave. The form of this correspondence depends on the nature of the language:

---

[3]This is the term we use for creating syntax-denoting operators from existing ones, it will be explained later.

[4]It can be claimed that there is only semantics and no evaluation, but again, this is an irrelevant discussion given our target languages as it only shifts terminology from 'evaluation' to a 'semantic function'.

[5]A more playful example from Smullyan: "This sentence is longer than 'this sentence'".

- In a natural language we want quoted text to be related to the actual meaning of that text; for example, the words "gray cat" denote a gray cat.
- In a programming language we want evaluation of quoted source code to behave in the same way as the code evaluates when unquoted, otherwise we have a plain interpreter/compiler for a different language. (Alternatively, if we reflect a different aspect like types or meta-objects, we want the semantics of these represented objects to be tightly related to the semantics of the objects they represent.)
- In a logical system, we want a reflection inference rule that can take a piece of quoted inference and if that inference is valid, conclude that the actual fact is true (*i.e.*, provability of some represented term implies that the term itself is true).

2.2 Reflection in Formal Languages

Formal Languages

When talking about formal languages, we have to be more precise than above. We begin with programming languages, and then discuss languages that are used in logic.

(Note that natural languages *may* be formalized — for example see Montague grammars [54]; it is just in the context of this work that we treat natural languages informally.)

A programming language has some formal rules for constructing its syntax, and a function that evaluates such input, producing some result. The operational semantics of the language is defined by this evaluation process that turns syntax into values. The evaluator can come in several forms such as an interpreter, or the composition of a compiler and machine execution. For the purpose of executing programs, a programming language is adequately defined by its syntax definition and its operational semantics.

Implementing reflection in a programming language context can mean different things depending on the reflected properties — we can reflect an object system creating a Meta Object Protocol (MOP) [45], we can reflect types [74], or we can reflect inherent properties of our language (like exposing unification in a Prolog-based languages). An interesting attempt at surveying some of these and relating them can be found at Demers&Malenfant [26]. In the context of this work, we focus on the traditional meaning of reflecting evaluation (*i.e.*, reflecting the operational semantics) — where reflecting a language means having the ability to write code that can generate and execute code in the same language.

Such reflection is, of course, almost always possible, since even with primitive language like assembly, we can use the operating system to write a text file containing some code, invoke the assembler over this file and execute the result. This is, however, an extremely crude way of implementing and using reflection because:

1. It relies on features that are external to the language itself (the operating system and accessibility of a compiler in this case) which is inefficient, might not be available at run-time, and does not explain how the semantic circularity is achieved. Moreover, this does not constitute a theoretical explanation of reflection which is our main focus.

2. Reflection is achieved through manipulation of code as flat strings, which is a low-level representation that is difficult to manage and understand, mainly because flat strings fail to represent the recursive nature of the syntax [9]. An even more extreme example of this is Gödel numbers [32] which can theoretically be used like any other encoding, but are not intended for practical use.

Smullyan's "Diagonalization and Self-Reference" [67] provides an excellent and very extensive introduction/survey of self-reference, and various quotation schemes that make it possible, though it does not address issues that arise in computer-implementations.

For the purpose of reflecting syntax, what we therefore need is some data structures *within* the language that can represent syntax, and, some mechanism to specify such quotations. *Quotation* of a piece of syntax S_0 in this context means finding a piece of syntax S_1 that *evaluates* to an object which is a representation of S_0. This is similar to the natural language case, where S_1 is related to S_0 through its operational semantics.

As mentioned earlier, the obvious way for representing syntax is to define recursive data structures (assuming the language has some way to define such structures). This can vary from verbose representations like the Abstract Syntax Tree entries used by CamlP4 [25] to the uniform lists of Scheme [44]. Note that these data structures participate in defining the line between syntax and semantics — everything that can be parsed to such structures is considered valid syntax.

The next step on the way to reflection is to have some form of an evaluator available at the user-level. One way of achieving this is to implement one — this has the advantage of requiring only user data structures and Turing-completeness (ensuring that a universal machine is possible in the language). An obvious reason for rejecting this is that it is basically re-inventing the *same* wheel you are already riding on, but an even stronger reason is that this is not true reflection in the sense that the implemented evaluator has no relation to the language used except for the programmer's wishful thinking. As a design principle, "true" reflection should be enabled using the actual evaluation function which executes the program itself — this is by means of exposing it to the language, making *both the meta and the object levels share functionality*. This *guarantees* the third property mentioned above (the correspondence between the language and the reflected language). This is the major motivation behind this work and the main thread of discussion, and will be further discussed in Section 2.5.

Formal Logical Languages

Implemented logical languages are formal, which makes their study similar to that of programming languages. However, the common approach in logic is treating the actual language syntax as secondary to the main issue of *truth* — many times not even completely specified[6], since it seems that the only hard requirement is being able to communicate logical content with people. This difference comes from a fundamental difference between the logic and the programming languages communities, where each side has its own methodologies and goals. As we shall see later on, this difference leads to very different approaches in reflecting a theorem prover system. Harrison [35] gives a survey and a critique of reflection in theorem proving; in this text he questions the practical necessity of reflection, which is perhaps another factor contributing to this methodology difference. Personally, I believe in a strong relation between programming languages and theorem proving, therefore I think that given the extreme usefulness of reflection in the first, it is likely to be just as useful for the second — providing the same "leap in expressive power" (borrowing the words of Taha [70]).

Implementing reflection in a theorem prover does require a complete and precise syntax definition because (a) theorem provers combine logic with a computer implementation therefore the syntax used to communicate with the machine needs to be specified, and (b) reflecting syntax turns it from a second-class tool that is used to describe concepts to a first-class semantic object type.

The semantics of logical systems are quite different from those of programming languages. Operational semantics are sufficient for programming languages as they are mostly concerned with program execution, but a logic has truth semantics. For example, reflecting a PL's operational semantics is usually theoretically easy, but reflection of the truth semantics of logic is harder (the former concerns a universal machine which is always possible with TM-equivalent languages, and the latter is usually impossible (*e.g.*, Tarski's theorem about the undefinability of truth.)). However, the situation is still quite similar to programming languages: truth semantics includes some object domain, and to reflect the language, the first thing we need are values that represent syntax and the syntax for denoting such values. The next step is, again, reflecting some semantic property, and there are several options for this, but usually the provability of logical sentences is the property we want to reflect.

In a theoretical logical world, the objects that represent syntax are as abstract as any other concept such as numbers. Any such representation would suffice, since there is no universal symbolism that can be used for literal quotations. This means that the preferred choice would be the one that is simple to handle theoretically. This is the reason Gödel numbers make sense in such a logic — the only thing they require is being able to represent integers, which is available in most 'interesting'

[6]Issues like renaming bindings for substitutions or specifying how multi-variable substitutions are made are often hand-waved.

logics. However, this requires tedious maintenance of translations between numbers and the syntax they represent, resulting in complex syntactic functionality[7]. High-level operations like alpha-equality are extremely hard to use when such a representation is employed, even if implemented as a computer system. One of the outcomes of this is that actual Gödel numbers (beyond a demonstrational toy example) are rarely seen or used.

In contrast to that, a theorem prover that implements a logic is a computer program, and as such it already contains syntactic data structures and functionality to deal with them. These syntactic data structures are at the opposite end of the usability scale when compare to Gödel numbers — not only are actual structures used, they are the only way to communicate with the machine. What this means is that the actual syntax that is used to communicate with the theorem prover is already made of object level values of the implementation (= the meta-level) — so we do have 'real symbols' that can be quoted, by making the *same* syntactic objects be available as semantic values (with some quotation syntax to denote these values). A considerable advantage of this method is that most of the tedious syntax functionality is eliminated since it is part of the implementation, and we will see additional advantages in Section 2.5. The disadvantage is that it requires a logical account for something that used to be a mere implementation issue, unlike, for example, integers that precede the computer implementation and therefore are always formalized. Note, however, that any theorem prover system will have some form of syntax which can be reflected into the system — and the theoretical account of this fact will demonstrate how this would be possible even in an 'on-paper' system (where an exact 'implementation' is left vague).

It should be mentioned that the concept of a meta-language is not at all foreign to theorem provers – in fact, a programming language that grew out of the theorem prover world is ML ("Meta Language") [52]. This language was designed as the meta-language for the LCF theorem prover [33], and later on became the popular choice for theorem provers at large, including Nuprl and MetaPRL [16, 36]. However, conventional use of ML in a theorem prover does not make it reflective: this meta language is used as an implementation language, providing means to define syntax (*e.g.*, terms as a datatype of ML) and proofs (same for proofs), but it is not accessible from within the logic itself.

2.3 Case Study: (Pure) Scheme and Nuprl

We now compare some of the relevant features of two formal systems — Scheme and Nuprl. The comparison is made to motivate our notion of shared functionality between the object and the meta levels, which is an issue that is approached differently by the two communities. This difference is a bit surprising given that the two sides are intimately tied together: Scheme is an untyped functional pro-

[7] Gödel numbers are essentially flat character strings, encoded with primes — there is no recursive structure beyond the induction principle.

gramming language that is inspired by the untyped lambda calculus which Nuprl contains, both Scheme and Nuprl rely on a very uniform syntax structure, which is manipulated directly (programs are S-expressions in the former, a structure editor is used in the latter).

It is interesting to note that both communities owe a great deal to an influential logician — Alonzo Church. Church's introduced lambda-definable functions [12, 13] as a new basis for the foundations of mathematics, instead of set-theory-based functions that were more common at the time. Later on, Church contributed to making type theory accessible [14], and to the Lambda Calculus [15] which is a direct ancestor to both Scheme and Lisp, and his work contributed to making type theory accessible. The two directions have been developed in separate ways, but in the last decade or so, the gap has been constantly shrinking with programming languages research becoming more formal (*e.g.*, the propositions-as-types principle [21, 40, 24, 48] is now common practice), and theorem provers popularizing logics that are tightly related to computations and programming languages [5, 18, 56, 17, 49, 61].

The reason for our choice of Nuprl and Scheme is not arbitrary: making Nuprl a reflective system is the main focus of this work, and Scheme is a good source of inspiration — the Scheme (and Lisp) community has been (and still is) dealing with reflection for decades. Scheme is particularly a good choice due to its simplicity compared to other languages, especially when it comes to its reflective capabilities[8]. Furthermore, we restrict the discussion to a purely functional subset of a typical Scheme implementation, side effects and other irrelevant concepts are ignored as well as implementations that diverge considerably from Lisp on syntax representation.

2.3.1 Syntax

Scheme

Scheme's syntax is essentially the same as that of other languages in the Lisp family. It is extremely simple — everything is either an atom of some fundamental type (*e.g.*, numbers and symbols), or a list of objects represented by some parenthesized whitespace-delimited sequence of objects. This is actually the syntax for general Scheme objects; the syntax for the language is a subset of these expressions — symbols represent variable references, lists represent procedure applications, a list beginning with the symbol '`lambda`' represents a function, etc. This is the first of several features that make reflection an integral part of the language. Quoting the Scheme Revised[5] Report [44, p. 3]:

[8]Note that there are some areas where Scheme is a little "less reflective" than Lisp, which is a source of religious wars between the two camps. For this work, it is important to use a simple language so Scheme was chosen — but we will not commit on following the standard definition but rather focus on common implementation techniques which are essentially the same as in Lisp.

> Scheme, like most dialects of Lisp, employs a fully parenthesized prefix notation for programs and (other) data; the grammar of Scheme generates a sublanguage of the language used for data. An important consequence of this simple, uniform representation is the susceptibility of Scheme programs and data to uniform treatment by other Scheme programs. For example, the '`eval`' procedure evaluates a Scheme program expressed as data.

Scheme implementations have a *reader* function ('`read`') that parses input, and a *printer* function ('`write`') to display values. The philosophy behind this is that printed output always represents values equal (modulo object identity) to the result of feeding this output back to the reader[9].

In Nuprl, the situation is similar — `terms` are the fundamental syntactic objects: they are used for input, output, and all internal processing. The information that terms represent comes from their operator name, their tree structure, and from attached atomic values (parameters). The structure of a typical Nuprl term is a tree structure of terms with no parameters and terms with parameters and no subterms as leaves. The equivalents of the reader and printer functions in Nuprl are its structure editor, used to enter terms, and a display-form mechanism that renders terms using plain mathematical symbols. This places Nuprl well among other "language-oriented" tools, including Scheme and Lisp environments that can be extended to new syntaxes, and modern frameworks like Intentional Programming [63].

2.3.2 Semantic Values

Values that are used by an implementation of our Scheme subset are of three major kinds:

- atomic values such as symbols, numbers and strings,
- composite values — lists holding an ordered sequence of values[10],
- function values, which can be generated by evaluating '`lambda`' forms.

Lists are implemented using the '`cons`' function that constructs a pair (head and tail) in memory (a *cons cell*) and the empty list ('`'()`'). The '`list`' function is a convenient shorthand for creating lists: '`(cons` x `(cons` y `'()))` = `(list` x y`)`'.

[9] This is not possible with all objects, for example, functions usually cannot be printed. Also note that feeding such output back to the interpreter will re-evaluate it unless it is quoted, which is an issue when printing datums that are themselves valid programs.

[10] We ignore other composite values like vectors and "dotted-lists".

The way syntax is represented in Scheme raises a subtle point: the Scheme interpreter sees all input through the glasses of its reader function[11] — so when Scheme source code contains, for example, a number, the reader will parse this and create the internal representation of that number, which becomes part of the (parsed) input source; therefore, the syntax for a number is *itself*, and there is no distinct 'numeral' type. Other values, including lists are also represented by themselves using the same mechanism, making it possible to use the language as a meta-level tool. Function values are closures, and have no external representation as they close over run-time information (the lexical environment).

As said above, the first step in achieving reflection should be extending the domain of the language so it holds *syntactic structure* objects. In Scheme this is done by making objects be the syntax that *represent themselves*, so the domain of Scheme objects is a superset of the domain of Scheme syntax structures. This point is unique to Lisp dialects due to the combination of an interpreted environment with the way syntax is represented as values.

Nuprl

In Nuprl, the language is even more uniform than Scheme: the only syntactic objects are terms. A term can have a list of (typed) parameter values, and it can have bound subterms. There are no distinct atomic values: they are replaced by terms that carry information in their parameter list. As in Scheme, all terms are subject to evaluation — canonical terms evaluate to themselves (also called value terms) and non-canonical terms have evaluation fragments that dictate how they are reduced. As an example, the equivalent of the Scheme atom '`3`' is a '`natnum`' term with a number parameter '`3`' and no subterms — it is a canonical term, so it evaluates to itself.

Similar to the Scheme case, the same first step that should be done to achieve reflection in Nuprl is making the syntax that the system uses available within the system. This can be done in several ways (see Section 3.3), but there is one important decision that should be made:

- We can use user-level Nuprl constructs to describe system values — making a user-level imitation of the implicit meta-level implementation.
- Or we can choose to somehow make the actual values that are used in the implementation available to users, achieving a similar situation to Scheme, where the implementation and the user *share* syntactic data structures as is the case with Scheme integers.

This decision, whether to imitate or to expose the implementation, is the main subject of Section 2.5.

[11] It is sometimes argued that Scheme is defined in terms of character strings — this is irrelevant here since we are interested in a typical implementation rather than the proper standard.

2.3.3 Evaluation

Scheme

A Scheme interpreter is basically a `read-eval-print` loop ("REPL"). The '`read`' and '`print`' parts are responsible for user interaction (mapping between internal objects and textual representations) and '`eval`' is the Scheme evaluation function, implementing its operational semantics. '`eval`' is a [partial] function that takes some input source code (an internal representation built by '`read`') and produces the results that this code evaluates to, if any. It is an applicative-order evaluator that uses lexical scoping.

The fact that '`eval`' is a function from Scheme values (syntax representations) to Scheme values (output values) might sound confusing at first: how can it distinguish values that *represent* code from other values? The solution is simple — the input is always taken as representing code and the output is always the resulting data. For example, if the code '`(list '+ 1 2)`' is evaluated, the return value is a list holding the symbol '`+`', and the numbers '`1`' and '`2`', and this is not evaluated further. In fact, if the '`eval`' function was not available to the user, then there was no way that it would ever get any input syntax other than user code — that is, it could never be applied on expressions that are results, losing the relation between Scheme syntax and values.

When a Scheme evaluation result is fed back into the interpreter, we get an equal object in the case of a non-symbol atomic value, but other values (lists and symbols) have meaning as syntax, leading to a different result. In other words, Scheme's evaluator is *not* idempotent. For this reason, the DrScheme pedagogic environment [29] helps beginner-level students getting used to the language by a customized printer function that displays such values as self-evaluating, making it *appear* idempotent. For example, the result of evaluating '`(list '+ 1 2)`' is the list holding the '`+`' symbol and two numbers, and it is printed as '`(list '+ 1 2)`' or as '`'(+ 1 2)`' by DrScheme. A few other Scheme implementations print values in a similar way, trying to avoid the inherent confusion.

The operation of '`eval`' on a given argument can be summarized as follows:

1. If the argument is a symbol, its binding in the current lexical environment is returned;

2. If it is any other atomic value, then this value is returned;

3. If it is a list and its first element is a special-form then the corresponding special evaluation rule is invoked;

4. If it is a list and its first element is a macro symbol, then the macro is applied to the *syntax* of the arguments (source code values) and the result is evaluated further[12];

[12]This is somewhat simplified: macros are usually expanded before evaluation.

5. Otherwise, the elements of the list are evaluated, and the first value is applied to the rest.

Nuprl

In Nuprl, terms are used as the elementary data objects, representing logical sentences, values, types and so on. In addition, the system contains an evaluator component that uses terms as an untyped lambda-calculus language. Some terms have an associated *evaluation fragment* which are functions that define reduction rules for terms. This evaluator is different than Scheme in that it is a normalizing evaluator: a term is repeatedly reduced until a canonical result is reached[13]. It is also different from Scheme evaluation by using a lazy head-first reduction[14]. This goes well with the semantics used by Nuprl — if a term x can be reduced to a term y, then it is always possible to substitute x by y. To summarize, the Nuprl evaluator is a lazy normalization (idempotent) function, mapping terms to terms.

Nuprl's approach allows a lot of freedom in the sense that different evaluation techniques can be intermixed, making it possible to reduce arbitrary subterms. But, as we will see next, this complicates making the system's syntax available at the object-level in a direct way.

2.3.4 Quotations (Representations)

Scheme

Scheme syntax is defined so that it is made of values that are part of the language, accessible to user code. The same holds for most other Lisp dialects. This, however, was not always the case: the Lisp 1.5 Programmers Manual [50] specifies two ways of expressing programs:

S-expressions These are *symbolic* expressions that are used for representing arbitrary data, including Lisp source represented in internal form.

M-expressions The actual source language that a Lisp programmer uses is named the "meta-language", since it specifies how S-expressions are processed. M-expressions can be represented in the form of S-expression for Lisp programs that use other Lisp programs as data.

The distinction between the two was supposed to be clear: programs in the form of M-expressions are what users use to write code for the compiler/evaluator, while S-expressions are used for internal data, representing parsed input as well as other user data. However, an evaluator function was written, essentially implementing a Lisp interpreter for Lisp programs, using Lisp S-expression data as input. This

[13]This is similar to the 3Lisp evaluator [65] that differentiates syntax values.

[14]This is, however, only the default behavior: terms can be reduced in any arbitrary order

led to the representation of Lisp code using S-expressions being the dominant programming language [51]. It might be possible to use a more 'standard' syntax in a smarter way than the one intended to be used in Lisp 1.5 — modify the reader and the writer functions so *both* use the same syntax — essentially modifying the way lists are represented as text using a display-form-like mechanism. However, this would require extra information such as annotating some symbols as infix operators, dealing with precedences etc, and this is further complicated by the fact that some lists are actually data. It seems like trying to separate Lisp syntax from Lisp data using such shallow approaches is impractical: such a separation will inevitably lead to a different language[15].

As said above, Scheme values represent themselves, and composite pieces of syntax are represented by lists, so quotation becomes trivial: to quote a piece of input source you simply write an expression that will have it as its result. The only piece missing from this picture is some way for writing expressions that evaluate to symbol values — the evaluator treats symbols as variable references, so a new special form is added to the language. This form, '`quote`', stops evaluation of a symbol: the result of evaluating '`(quote a)`' is the symbol '`a`'.

Now we know that:

- to quote a symbol, we wrap it by a '`quote`' special form;
- to quote any other atomic value, we simply use it (it is self-evaluating);
- to quote a composite syntax (a list) — use the '`list`' function to create the list out of expressions that evaluate to the content of the desired list.

For example, the quotation of '`(+ 1 2)`' is '`(list (quote +) 1 2)`'. The main problem with this is that it is not a *proper quotation* mechanism — the quotation does not directly use the quoted syntax. For example, quoting '`(+ 1 2)`' twice yields '`(list (quote list) (list (quote quote) (quote +)) 1 2)`'. The '`quote`' special form is therefore extended to stop evaluation of any syntax object, including lists. For example, the quotation of the above expression is '`(quote (+ 1 2))`' and the second quotation is '`(quote (quote (+ 1 2)))`'. This makes it a proper quotation mechanism using a special quoted context where code represents code rather than normal computations, just like quotation symbols in written natural language texts.

Nuprl

As mentioned above, the Nuprl evaluation process is designed to follow its intended semantics — so terms can be substituted no matter where they occur. Therefore, adding a Scheme-like '`quote`' term that creates a non-evaluating context is impossible, as it would require changing the Nuprl evaluator and its semantics in an incompatible way. There are other possibilities, which will be discussed in more

[15]The Dylan programming language is another indication of this.

details in Section 3.3. One thing to note is that the conventional approach is to use a simple recursive type definition (in the Nuprl case, this means a triplet of an operator name, a list of parameters, and a list of subterms), but as we will see, this is an undesirable solution.

2.3.5 Reflection

Scheme

Lisp was designed to be well suited for symbolic processing, including the processing involved in executing Lisp programs. Scheme inherited most of this design, and had '`eval`' added in its 5th revised report [44] in 1988[16]. Reflection is closely related to the sharing of syntactic structures between the implementation and the object levels, which is evident also in macros as source-to-source transformer functions. The combination of sharing syntactic structures and an evaluation function creates a simple reflected environment. The question is — how can a *user-level* '`eval`' function be implemented in Scheme?

As said in Section 2.2, we can get the evaluation function for a language that we want to reflect either via re-implementation or by exposing the actual evaluation function to the user level — breaking the abstraction barrier between the language and its implementation. Scheme implementations usually choose the latter: the '`eval`' function is the same as the one that the implementation uses for evaluating all code. This is simpler, safer and more efficient, as we shall see in Section 2.5. It should be noted that such an implementation is not required by the Scheme Report, the only thing that is required is the availability of an '`eval`' function that evaluates Scheme expressions.

Making the interpreter share syntax with user level programs is another example of such exposure — internal implementation functionality uses data structures, which are also reified as user-available data. The Scheme report requires that Scheme code is be represented as Scheme data, and exposing these internal structures is the most obvious approach. This way of reflecting a system by exposing some of its internal functionality is the main topic of Smith's work [65, 64] and was referred to as *procedural reflection*, we will generalize this idea as *direct reflection*.

Nuprl

Reflecting Nuprl should in theory be possible with any quotation mechanism. Previous efforts [3] (also [47, 41]) used the recursive data type approach, but this turned out to be impractical — it has led to re-implementation of large parts of the system in a formal setting, which grew too big to be easily manageable, and was never finished. The main concrete result appeared in [6], where reflecting

[16] Although it was always common for Scheme implementations to have an '`eval`' function.

syntax was not detailed. This work began as an attempt to achieve better syntax reflection which should eventually lead to a practical reflection implementation.

2.4 Quasi-Quotations

An issue that was not raised so far is that of *quasi quotations*. The problem with proper quotations is that it is only possible to quote pieces of syntax as complete units, not providing any ability to intermix quoted syntax and descriptions that refer to quoted syntax. For example, in natural languages there is no mechanism that provides an escape from a quotation — so it is only possible to have pieces of completely quoted text, or resort to tricks like:

- *He said: "send it to"—you-know-who—"please".*
- *He said: "send it to X please", where 'X' is you-know-who.*

and using Scheme notation for these:

```
(append (quote (send it to)) you-know-who (quote (please)))
(subst you-know-who (quote X) (quote (send it to X please)))
```

Note that in both cases, these sentences require access to syntax-manipulating functionality such as '`append`' and '`subst`' ("—" and "where..." resp. in the natural language case).

The problem of mixing literal quotations with descriptions is one that many logicians faced, most notably Quine [60]. Quine's solution is to reserve several Greek letters as *meta-variables*, making it possible to 'poke holes' in a literal quotation, making it a predecessor to quasi-quoting. The modern solution to this problem is a successor to Quine quotes: *quasi-quoting* — quotations that are "mostly-literal". These quotations are special in allowing a temporary escape out of the quoted context back into the normal language, where we can talk *about* syntax.

Quasi-quotations have become the standard tool that is used for mixing literal quotations and descriptions (*e.g.*, [22, 8, 53]). They are particularly common in programming languages — with Scheme and Lisp being the most obvious representatives, but used by practically any language with reflective capabilities such as CamlP4 [25] and MetaML [69]. Lisp is a pioneer in this area, since it needed some facility when macros were introduced into the language by Timothy Hart [43].

Using a quasi-quotes notation, the above is written more easily, since we no longer require syntax-manipulations, we simply escape back to the unquoted level:

- *He said: ⌜send it to ⌞you-know-who⌟ please⌝.*

and in Scheme:

```
(quasiquote (send it to (unquote you-know-who) please))
```

or using the common, more succinct form:

```
`(send it to ,you-know-who please)
```

The treatment of syntax in Scheme tends to be more complicated than simple lists of symbols, so sometimes '`quasisyntax`' and '`unsyntax`' are used; this is a fact that we mostly ignore throughout this text. A much more extensive discussion on this, as well as additional information on quasi-quoting (and quoting in general) can be found in Taha's dissertation [70].

Quasi-quoting is, of course, not an issue when proper quotations are not used. For example, if there was no generic 'evaluation-stopping' context in Scheme (if '`quote`' can only be used for symbols), then we would be forced to write:

```
(list (quote send) (quote it) (quote to)
      you-know-who (quote please))
```

in any case. In fact, the '`quasiquote`' functionality in Scheme is usually implemented through preprocessing — as a macro that converts its contents to usages of '`quote`' and other syntax (s-expression) manipulating functions.

Being able to mix quotations with free descriptions is very useful in practice — it allows combining the simplicity of plain quotations with the power of general descriptions. In any practical context, there is therefore a need for quasi-quoting, otherwise we end up with an environment that is either too restricted (allowing only literal quotations), or one that is too complex to use (using syntax operations). Regardless of the quotation mechanism we choose, we need to be able to use quasi-quotations or some equivalent mechanism. The question is whether we can have a simple yet elegant solution that provides this feature. As we shall see in Section 3.3.6, there is always a way to achieve similar functionality, even without proper quotations.

2.5 Exposing Internals vs. Re-Implementation

We now get to the major theme of this work — how should a computer system[17] be reflected? As we have seen, there are two possible high-level answers for this question:

- We have a system that can manifestly implement itself correctly, use this ability so that the external system reflects the internal one.
- The computer system is already implemented correctly, expose this implementation to itself, exposing the correctness argument (in a system that has a notion of correctness).

[17] One with some linguistic features, like a programming language or an implemented logic system.

2.5.1 Re-Implementation

If we follow the first solution, we get a re-implementation of the system within itself. There are a few problems with this approach. First, consider the involved effort: we are working *within* some computer system — a programming language, a theorem prover, an object system, etc — and we are encoding functionality that already exists as part of the system that we use. For example, if we implement a Scheme interpreter in Scheme, we invest our efforts in recoding functionality that is already part our implementation.

A justification for choosing to re-implement a system this way is that it provides a better explanation. For example, any implementation of Scheme is in fact a precise (Scheme) explanation of its operational semantics; a meta-object protocol explains a class system using classes. If the implementation is written in a different language, say C, then it already serves as a "C explanation" — but this provides an inferior explanation of the environment we work with, and it's not really relevant for this work since it is not reflection. Similarly, Nuprl is a big computer system that contains lots of functionality — but this is only a computer program, therefore it is only an operational explanation (only the "how", not the "what"). If we re-implement parts of Nuprl within itself we could get a much better explanation — for example, we could extract an alpha-renaming algorithm from a correctness proof, and reason about the involved term types.

But there is another, more serious problem: such a re-implementation has no relation to the original system[18]. For example, a Scheme meta-interpreter can be written in Scheme, but the result is not necessarily related to the original — we wish for the two language levels to be *identical*, but in practice there is nothing that guarantees this. This is desired even in case there are bugs involved: a buggy explanation of a correct system or a correct explanation using a buggy system seem useless. We therefore *want* a way to guarantee that possible bugs in the system are *necessarily* reflected in the explanation, which means that verifying either one is sufficient to know that the other is correct. To verify that the effort involved in a system re-implementation is correct, we therefore need more efforts in proving that it is indeed equivalent to the original one.

This proof is yet another possible source of errors. Finally, even if one goes to the trouble of providing such a proof, the result is still not robust — computer systems are dynamic objects: bugs get fixed, features get added and removed, interfaces change. This applies to practical systems like Nuprl, and a re-implementation will unavoidably lead to higher costs as the reflected version will require maintenance to for it to be synchronized with the actual one — such maintenance can be automated, but this too requires non-trivial efforts.

With some systems, it can be possible to rectify this problem by bootstrapping: when the re-implementation is fully functional, we use it instead of the original

[18]Unless extraordinary efforts are taken to keep the implementation and the re-implementation are synchronized.

one. This means that the 'real' system we work with is a *fixpoint* of its description. Many compilers, for example GCC and OCaml, are built this way. However, in the case of a theorem prover, this approach requires a prover that can prove itself correct and be able to extract its own code from this proof. This is quite difficult as theorem provers use truth semantics rather than the operational semantics of programming languages, and would require additional research.

2.5.2 Exposing Internals

The alternative to re-implementing existing functionality is, obviously, re-using it. This usually implies breaking some abstraction barrier — for example, in CLOS [45], the Meta Object Protocol is deriving its power from the fact that the object system is completely exposed: details that are usually buried deep inside implementations of other object systems such as class definitions and method dispatch, are defined in CLOS in terms of the object system itself, allowing definitions of different kinds of classes and generic functions. The abstraction barrier of standard object systems is a black box that defines how the system behaves, but in CLOS it is broken in the sense that it is no longer a black box — it is an open one, with parts of the implementation exposed in a well-documented way (the various protocols). In fact, *breaking the black-box interface to an existing system is the essence of reflection* — the result is by its nature a system that *can access its own internals*.

Exposing a system's internals to make it reflected requires a design decision: the reflected interface can have multiple forms with varying degrees of openness. For example, Kiczales and Paepcke [34] demonstrate three levels of reflecting an object system — allowing only introspection of internal data, adding invocation of internal functionality, and finally making internal functionality modifiable: intercession. In the case of reflecting Nuprl's syntax, we can either expose the internal term language interface in its full power, or we can expose a higher-level view such an "alpha-equality" interface which abstracts away these names[19].

Implementing reflection through internal functionality does not suffer from the above mentioned disadvantages of a re-implementation. The first big win in this case is a major one: there is no need in any proof of equality of the two levels as they are *identical*. There is no need for a fixpoint since we use the implicit circularity achieved by the ability to expose system functionality to its object level. In a programming language, this means that we immediately know that the reflected object-level language is *inherently* identical to the meta-level one. The result is also robust in the sense that changes to the implementation are also changes to the reflected system. We name reflection implemented in this way *direct reflection*.

The second advantage is that the involved work is minimized to parts that are actually needed — parts that did not already exist. In the case of reflecting an

[19] As we will see in Chapter 3, exposing terms as accessible object-level values leads to semantical requirements that restricts this choice.

evaluation function, the only required work is interfacing the internal function to the object level, and in the case of reflecting syntactic functionality in a theorem prover the additional work is in expressing facts that the operational implementation lacks (which can still be substantial). The fact that almost no additional work is needed for reflecting evaluation makes exposure a natural implementation technique — doing it any other way (*i.e.*, through a meta-circular evaluator) seems pointless for any purpose other than a pedagogical one. This is evident in informal discussions too: if we have spent a book chapter defining computation using Turing Machines, then when we get to talking about a universal machine we simply say that it is a Turing Machine that follows the rules in the previous chapter rather than repeat the description in the form of a fully-specified machine implementation. Note that it is not always the case that exposing functionality is easier: for example, reflecting an object system involves more work — the internal implementation is usually not defined in terms of itself, which means that exposing functionality requires more work (usually rethinking the implementation), and might also lead to performance costs. As a result, the conventional approach to reflecting an object system (*e.g.*, Java and C#) tend to be restricted to introspection only — where meta-entities such as classes and generics can be inspected via the object system, unlike the ambitious CLOS definition that allows intervening in the meta protocol.

In this work we conjecture that theorem provers tend to suffer a similar problem. The logical language is quite different from its implementation, and in addition the computer system is theoretically irrelevant to the proofs it deals with. Therefore, there is a fundamental methodology difference from the world of programming languages: the implementation is considered merely as a tool that is to be ignored for the real work (it gets the same importance pencils get when dealing with logic). As a result conventional reflective theorem prover implementations, such as Aitken's [3], have usually been attempted via re-implementation[20]. In this context, even Gödel numbers appear as a viable option for reflecting syntax, although they are impractical as a basis for an implementation. The nature of reflection is that any description, including a theoretical one, becomes part of the described system, falsifying the intuition that the system is an abstract ignorable entity (thus, re-implementations must develop the functionality of a complete system). Aitken's work [3] is a demonstration of this, resulting in extensive efforts. (Note that this conjecture is specific to theorem provers; in a theoretical logic context, intensionality is similar to our direct reflection [66].)

A few more issues play a role at choosing an implementation strategy:

Additional work In the context of reflecting syntax in a theorem prover, the gap that needs bridging is more substantial than a simple object-level interface to an evaluation function. The theorem prover deals with truth semantics, which means that facts *about* the behavior of its internals do not exist in the

[20]The Boyer-Moore system [11] is a little different in being an extension of Lisp, therefore inheriting some of its linguistic features.

operational semantics of its implementation. Therefore, these nonexistent facts cannot be exposed, which translates to necessary work.

Type-system independence The reflected layer must be expressed in the theorem prover, using any tools it can offer us. If we choose a re-implementation, then we need a system that can deal with certain types such as symbols, tuples, and lists, and related functionality like structural induction etc. This might not be available in a theorem prover, or available in a few different forms in a framework that can have several alternatives. In contrast, exposing internal functionality must take the form of primitive functionality — a new primitive data type that is the reflection of the internal type.

Decreased flexibility Another point to consider is that exposing the internal syntax to the object level provides a tighter connection to the specific implementation, makes it less viable to explore different approaches to representing syntax since a different implementation implies changing both the meta- and the object-level.

Decreased portability Finally, although the principle of exposing internals can apply to any computer system, features of the resulting reflected system are unique to the system as they depend on features of the underlying implementation. For example, while MetaPRL [36] uses a term structure that is very similar to Nuprl terms, the implementation is quite different, therefore achieving reflection via exposing of these internal structures will result in two different systems — this is evident in the form of reflection that was used in MetaPRL [55]. In other words, except for the implementation techniques we use, the portability of the solution is mostly lost.

2.5.3 Duplicating Information Considered Harmful

As said above, the main advantage of re-implementation is a pedagogical one — reflecting the existing implementation in a better language, leading to a better explanation. In programming languages a better explanation means one that is written in the same language, and in a theorem prover it means providing a factual explanation — using truth semantics. However, in my judgment and experience, the additional massive effort required for a *practical* implementation outweigh the pedagogical advantage. By exposing internal functionality we make it possible to use just parts of it, re-implementing others when there is a pedagogical need.

When considering the pros and cons of implementing reflection using either re-implementation or exposing, I conjecture that exposing is a better approach, but this might be a result of personal bias: it is my belief that the holy grail of computer science is the concept of abstraction — and the essence of abstraction is making it possible to put an identity on a piece of information, and re-use it multiple times rather than duplicate it. This applies to any kind of information in general, and to algorithms in particular. I think that this is the root of the problem

of the impracticality of previous attempts to reach a practical implementation of a reflected system. This summarizes the main principle of this work, and will be used to guide all design decisions:

> *to gain the full power of reflection, internal meta-level functionality is exposed as object-level functionality.*

Note that when we get to design the actual implementation, it could consist of a mixture of exposures and re-implementations. The system we want to reflect will usually consist of several components, each one can be implemented differently. Our design principle will therefore be relevant in several occasions.

Chapter 3
Implementing Quotation

In the previous chapter, we saw that the first step in implementing reflection for Nuprl is to reflect its syntax. This is the foundation for the rest of this work, and will have a high impact on the reflected system. In this chapter we first describe Nuprl terms, then discuss the various options we have for reflecting syntax, and finally choose one and describe the implementation. Following the main principle laid out in the previous chapter we aim at reflecting syntax through exposure of implementation details. A consequence of this is that implementation issues cannot be dismissed as "technical details" — they play an important role and are central to the design of the reflected system — such issues come up when we expect to expose implementation details to user code.

But before we begin, we introduce a syntactic notation that will be used throughout this text.

3.1 Syntactic Sequence Notation

This work is centered around syntax, both in the theoretical and the practical discussions. Unsurprisingly, our syntax contains many forms of *sequences* (*e.g.*, binding lists, subterms, sequent hypotheses), which are used in both the meta-level (*e.g.*, rules) and the reflected object-level. To make this more convenient, we adopt Scheme's *ellipsis notation* [44] for specifying sequences: an ellipsis ('...') specifies that the preceding syntactical unit denotes a sequence — when this syntactical unit contains a meta-variable, then this variable actually stands for a sequence of the matching variables indexed as usual (or with multiple indexes in case of nested ellipses); when the number of syntactical elements in the sequence is significant, we will subscript the ellipsis by it. For example, we can say that '$\lambda\langle x, ...\_n\rangle\,.\,x_i$' is a λ-term that denotes the 'Π_i^n' projection function of n-tuples. In places where the sequence length is insignificant, it will be omitted. Similar notations are common in informal explanations, for example, an overbar notation for sequences is common in Nuprl-related literature.

Note that this is merely a notation — the actual representation that is used can vary. For example, the representation of Nuprl terms uses lists, but the theoretical account often uses indexed functions. The actual representation will not be ambiguous as it will always be explained in advance. In addition, we will use this notation to specify syntactic transformations (for example, to define rewrite rules) and for that it will be extended in Section 6.2.

3.2 Nuprl Terms

Nuprl terms are the fundamental data structures that are at the core of the system. They have the same role S-expressions have in Scheme — serving as a general-

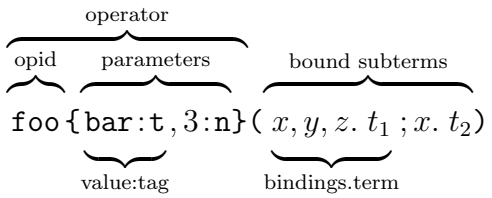


Figure 3.1: Term parts

purpose uniform data structure that is used to represent *all syntax* (expressions, propositions, types, programs, etc), as well as administrative system information (display forms, rules, editor information, etc). Unlike Scheme, they exist only at the meta-level, thus as part of the implementation.

This section is a quick informal summary of Nuprl's term structure, related functionality, and usage issues, with an emphasis on points related to reflecting the syntax. It is mostly a gloss of Aitken's description [3, Chapter 1] and the Nuprl Reference Manual [42, Chapter 4], which should be consulted for additional details.

3.2.1 Term Structure

Nuprl terms are defined as a recursive data type that has the following structure:

1. An *operator*, which is made of:
   - An *operator identifier*, abbreviated as "*opid*".
   - A sequence of *parameters*, each one is a pair of:
     - some value;
     - a type tag for this value: an *index family*.
2. A sequence of bound subterms, each one is a pair of:
   - a sequence of bound variable names (strings);
   - a term.

Figure 3.1 illustrates the various parts of a term.

For example, the λ-expression 'λ`x.x+1`' is specified by a Nuprl term that has the following full form:

`lambda{}(x.add{}(.variable{x:v}();.natural_number{1:n}()))`,

or, removing redundant punctuation we get

`lambda(x.add(variable{x:v};natural_number{1:n}))`.

Note that parameter values are part of the operator, so '`natural_number{1:n}`' and '`natural_number{2:n}`' (abbreviated as '`1`' and '`2`') are different terms because

they have different operators. Terms are actually distinguished by their signature: the *signature* of a term is the pair of its operator and its arity, where the *arity* of a term is defined as the list of its binding lengths.

Using the ellipsis notation, a term with k parameters and n subterms, each with m_i bindings, looks like '*opid*`{`*v*`:`*f*`,...`$_k$`}(`$x, \ldots_m . t; \ldots_n$`)`' where '*opid*`{`*v*`:`*f*`,...`$_k$`}`' is the operator. The signature of this term is therefore: '*opid*`{`*v*`:`*f*`,...`$_k$`}`$[m, \ldots_n]$'.

Operators are a discrete, *open-ended* set, which means that Nuprl terms are also open-ended. This is not only due to the nature of operator identifiers, but also due to parameter types. Whatever representation we choose for quotation of terms, we have to account for this open-endedness of terms, and allow for quoting arbitrary terms with arbitrary parameter kinds.

Bindings

As expected, bindings are a major issue when dealing with Nuprl's term language. Internally, the implementation simply uses strings for bindings: the list of bindings for each term's bound subterm is a list of strings, and a variable reference is a term that contains a string parameter value that names the variable. For example, in

`lambda(x.add(variable{x:v};natural_number{1:n}))`,

both occurrences of '`x`' are plain strings. The connection between binders and bound occurrences is made through the family of variable name parameters (the '`v`' tag), which holds string values comparable to binder strings.

To make interaction with the system easier, and texts such as this less verbose, variables and natural numbers are abbreviated as their value, *e.g.*:

`lambda(x.add(x;1))`.

Such abbreviations are ambiguous ('`foo`' can be either '`variable{foo:v}()`' or '`foo{}()`'), but in practice this will not be a problem.

Note that Nuprl uses terms-containing-bound-subterms as its basic objects, but in some situations it is more natural to use bound terms (containing bound subterms) as the basic type. These two approaches are quite similar, the main difference is whether binding names are associated with a subterm or with its surrounding term. Using terms is a little more natural for Nuprl, since the meaning of an operator depends on its arity — for example, it is more convenient to say that a λ-term has an operator of '`lambda{}`' and an arity of [1] instead of talking about all bound '`lambda{}`' terms that contain a bound subterm with a single binding. Another implication of this is that binders are taken as part of the surrounding syntax rather than part of their scope term. We will, however, occasionally switch to the alternative view in situations it is more convenient to use.

In Nuprl, the same binding name can occur multiple times as long as occurrences are in different subterms, which is a feature that should be available in the reflected syntax too. In the following theoretical account we forbid multiple occurrences of a name in substitutions; the reason for this is our commitment to follow the implementation, judging alternatives by this criteria renders most as irrelevant. The point of duplicate binding names demonstrates the advantage of

using direct reflection: there are many design decisions that were made during the implementation of Nuprl, some are hard to be aware of until we inspect the relevant low level details, or invest enough efforts in considering alternatives. No matter what these decisions are, we can be assured that we will reflect the actual functionality including such subtleties that we are not (yet) aware of, essentially trusting years of human efforts.

3.2.2 Interaction

Nuprl is an interactive system, that also has a compiled mode. It's interactive functionality (dealing with terms, editing proofs etc) is available at three levels. On the lowest level, the system is implemented in Lisp, making it possible to operate on Nuprl objects via Lisp. Since this is the implementation level, it is not used for normal editing. Nuprl includes an implementation of ML, which is the user-level language that is normally used by users when programmatic capabilities are needed. In fact, large portions of the system are written in ML. The ML level allows easy access to Nuprl objects, including types for all syntactic objects: it is this level of functionality that we wish to reflect.

Structure Editor

The third level is the interactive system. This level consists of several important parts. First of all, interacting with Nuprl involves lots of term editing — for this there is a structure editor that is used to enter terms. The editor itself is mostly implemented in ML, following simple specifications of common term structures. Using it is similar to typing Lisp S-expressions: the user types a known operator id (or a mnemonic), and the editor lays out the expected structure of the term with empty slots to be filled out next.

Display Forms

The second important part of the interactive system is its usage of *display forms* to present terms. The syntax structures underlying terms is a uniform representation that is convenient for a computer to work with, but humans prefer a more succinct and/or familiar presentation. For example, it is easier to read 'λ`x.x + 1`' than

`lambda{}(x.add(variable{x:v}();natural_number{1:n}()))`,

which is a more precise presentation of the term structure. This is especially important in the context of logic, where notation is important, and can vary according to context (the field conventions, the author's preference, etc.) following the need to interact with other people (including people who are not using Nuprl). As a demonstration of the need for display forms, the shorthand notations that are used in this text (*e.g.*, using numbers and variables as such (see Section 3.2.1)) can also be considered as a kind of display form.

Nuprl accommodates varying layouts by separating the internal syntax structure from its presentation form which is determined by a display form. In essence, it separates the abstract syntax trees that are use by the system to maintain the logical structure of terms and the presentation of this same syntactic information relying on social conventions (perhaps ambiguously). The logical structure remains faithful to an exact, machine-manageable form.

The display form specifies how terms are presented, which can include more than a simple textual rendering information. For example, it can contain precedence information which makes it possible to avoid some parenthesis, it can avoid showing some parts of the syntax, and it can even present a nested structure as a flat one (*e.g.*, 'λ`x.`λ`y.y`' is rendered as 'λ`x,y.y`') (this is a specific example where a particular rendition is chosen according to the term's context). Most display forms are specified in the Nuprl library, meaning that users can customize their display forms according to their personal taste and needs, yet still be able to interact seamlessly with people who expect different notations. Also note that the display form can go beyond simple text: they contain layout information (line breaks & indentation), they can be used to render complex layout (*e.g.*, via LaTeX) and textual attributes (*e.g.*, font attributes), and they can be extended to handle other arbitrary computer display properties. Finally, the structure editor maintains the relation between a term's rendered form and its underlying structure; this relation is used for operating on term parts and navigating between them.

This mechanism presents a unique opportunity: it can be used to enhance the presentation of quoted syntax so it is close to the rendition of the unquoted syntax. More on this below.

3.2.3 Substitution

The fundamental operation that builds on the concept of bindings and scope is substitution. It is probably the single most important concept when dealing with formal syntax, yet various texts ignore parts of its formalization, possibly handwaving "the right thing" instead. On the other hand, a running computer system is completely formal in the sense that an fully-specified algorithm is required to perform substitutions: for example, Nuprl has a substitution function that can simultaneously substitute multiple variables at the same time, a process that is rarely addressed properly in logic texts; we take such capabilities for granted since we use the system's functionality.

The actual Nuprl substitution algorithm is pretty complex due to the fact that when renaming is required, it happens in a way that matches user expectations for minimal renamings. The main reason for this is that it is an interactive theorem prover, and as such it is important that variables are renamed in a manner that humans can understand and even predict. Furthermore, this ability can be used (or abused) by ad-hoc mechanisms that rely on being able to relate a renamed variable with the original name. There are a few additional details that are ignored here: the bottom line is that *we take Nuprl's substitution algorithm as given*, and assume

it satisfies certain necessary properties.

3.2.4 Term Meaning: Evaluation

The meaning that Nuprl assigns to terms is related to the way the system evaluates them. The Nuprl type theory requires that terms are compositional: the meaning of a term depends on the meaning of its subterms rather than on their syntax. This has some rather important implications: (a) a term cannot have different meanings when it is a subterm of different context terms, (b) any subterm t of a term s can be substituted by any equal subterm t', and the result will still be equal to s. In addition, no special meaning can be given to binding names, so alpha renaming preserves meaning, in other words, *the Nuprl theory is defined over alpha equivalence classes*.

The Nuprl evaluator is lazy by default, although (following the above restriction), any subterm can be reduced at any time, which leads to a few additional evaluation schemes, as well as making it possible to implement any arbitrary-order evaluations. In any case, evaluation is always partial, as there are always well-typed terms that diverge.

Value, Abstract and Primitive Terms

We distinguish two basic types of terms: *value terms*, (also called *canonical terms*) are terms that stand for a value. Evaluating such a term simply yields itself as there is no reductions available. These terms can come in many forms: they can contain multiple subterms (*e.g.*, pairs), and they can use binders (*e.g.*, a '`lambda`' term).

Non-canonical terms are terms that can be further 'reduced', possibly leading to an infinite loop. Roughly speaking, there are two kinds of such terms: primitive and abstract. Primitive terms come with the system, and have some fixed assigned meaning; they are the primitive 'built-in' terms that the system comes with (and their implementation is part of it). Abstract terms are those that are defined as 'abstraction objects' in the library. These abstractions are simple (non-recursive) pattern definitions that specify what the abstract term is defined as — when evaluating a term, such abstractions can be *unfolded* to their defined form. The core of the system is made of a small number of primitive terms, and the bulk of terms that users encounter are defined as abstractions over these building blocks, defined in the standard library.

When Nuprl encounters a term that is not a primitive, and is not an abstraction, it treats it as a value term.

The Nuprl Library

The Nuprl library, which has been mentioned a few times above, is the main knowledge base with which users interact. The main core of the system imple-

ments essential functionality, and the rest of it is defined within the system. Nuprl comes with a *standard library* that contains most of its functionality. The library is divided into *theories*, each of these is a collection of abstract term and display form definitions, rules, theorems, ML code objects, etc. The library is therefore an ordered list of Nuprl objects: the system tracks object dependencies, and requires that objects depends only on preceding objects, avoiding mutually recursive definitions.

Users interact with the system via library objects, and their work is saved in a form of a theory (or several theories) which usually follow the standard libraries that come with the system. A theory is roughly equivalent to a source file that contains several related definitions. Substantial pieces of work that were implemented are available as additional collections of theories.

Note that Nuprl does not have a notion of a 'global environment' that binds free variables. Abstract terms are used instead, and they are not first class in the sense that the do not stand for functions that can be quantified over or bound in any other way. As a result of this, only closed terms can have a top-level meaning in Nuprl; open terms are nonsensical, they cannot be used in any form.

3.3 Syntax Representation Options

The crucial starting point of the reflection implementation is choosing an appropriate representation syntax. Back in Chapter 2 we went over various issues regarding different options for doing this.

First, let us remember the syntactic capabilities of Scheme which was mentioned in the previous chapter. Scheme has a particularly elegant way to reflect its syntax, which serves as a motivation for this work. This capability is evident in two important ways.

- First, Scheme's syntax objects are S-expressions, a common data type that is used by the language implementation as well as user code. The primitive functions that construct and operate on S-expressions, and availability of the basic building blocks of these recursive structures (symbols, numbers, string, etc) for users provide a simple but rich system that can be used to toy with language design[1].

- This functionality by itself makes code processing available, but Scheme goes further along and provides syntactic forms that make this even more convenient. Strictly speaking, S-expression operators and primitive syntax objects are the only thing that is required. Scheme could therefore be just as expressive if its '`quote`' form operates only on symbols, changing the quoted element

[1]In fact, a common criticism of the Scheme and the Lisp community is that "everyone is a language designer". This is leading to subtle but deep fragmentation that other language communities don't have as they have a much clearer line between the language designers and its users.

from a variable reference to its denoting syntax. However, Scheme extends '`quote`' so it can be used with any syntax, making it a literal quotation — making the Lisp family of languages the only one that has *proper quotations* available. This should not be confused with preprocessing mechanisms such as CamlP4 [25], which, as we shall see below, can be used as a substitute for a lack of proper quotations. Scheme goes on to include a '`quasiquote`' form that is similar to '`quote`' except that it is used with "almost-constant" literal quotations: where we want to quote a piece of code where some of its parts are conventional code description (using S-expression functionality to construct syntax).

We now consider some of the representation possibilities and related implementation techniques, bearing the Scheme model in mind as a good system that combines a simple-but-powerful syntax system with convenient user facilities that make it easily accessible.

3.3.1 Quotation Context

Indeed, an ideal syntax representation would be using *proper quotations*: where the quotation of some term t has t itself as a subterm. This is a natural representation, as witnessed by natural language quotations. In Scheme, it is implemented by the '`quote`' special form, which always evaluates to its contents — in effect, it is an "evaluation stopper" which turns its sub-expression from a syntactic value to a user value. The natural language analogy is obvious: when we use quoted text (for example "hello" in "I said 'hello'"), it has the exact same form as normal text, except that it appears in a quoted context that tells us that we should not use this text as denoting its usual meaning (a greeting), but as something that talks about the text itself (the word "hello").

However, this approach collides with the way Nuprl is implemented — as described in Section 3.2.4: in Nuprl, a subterm t that denotes some value can always be replaced by any other term that stands for the same value; computing a term is done through a normalization process, unlike conventional programming languages evaluators[2]. It might have been possible to rework the implementation and the semantics so a Scheme-like quotation context is used, but this would radically diverge from the current system. Changing the evaluator will lead to changes in the semantics, in the theory, and in the formal account of the system; this effectively requires reworking decades of work, moreover — it diverges from Computational Type Theory which has a longer history and deep historical roots. This approach is therefore disqualified as impractical.

[2]Nuprl's evaluator is somewhat similar to Haskell.

3.3.2 Black-Box Quotations

There is a slightly different way of achieving a quoting context in Nuprl without changing its semantics: a quotation can be a term that contains the quoted term as a parameter value rather than a subterm. This is easily achieved by extending the system by a new parameter kind (an index family). Parameter values are taken as part of the syntax, therefore they are not susceptible to evaluation — in fact, their role in the semantics is purely syntactic.

Using such a parameter kind, we can use '`quote`' terms which have the term they stand for as a parameter value. For example, to quote '`add(1;2)`', we use '`quote{add(1;2):t}()`'. This way, quotations are always atomic values — they contain some information in an opaque way, requiring new construction and deconstruction functions. In other words, the quoted term is represented in a way that is not much different than a string that contains the term (in some unambiguous way): the semantics of Nuprl involves terms that contain subterms; a term object that is included as a parameter value is no different than any other object that we happen to be able to map onto terms. Another (extreme) example of such a quoting mechanism is Gödel numbers: they are just as opaque, but are worse in that they require additional (and substantial) parsing work.

Ignoring the semantical issue, it seems like this approach does make it possible to have proper quotations in Nuprl, but there is an additional, more serious problem with it — the resulting mechanism is similar to the natural language quotations in that there is no way to mix literal quotations with descriptions. The only way to achieve some equivalent functionality would be similar to the way it could be done in natural language (see the example on page 23), using some substitution function and two literal quotations, or other similarly crude facilities. The reason for this difficulty is the same reason that made it possible to mention terms as proper quotations in the first place — the quoted terms appear in a place that does not have terms currently, which means that such an implementation technique is bound to expose quotations as mere black-box values.

Being able to mix descriptions and quotations is a crucial property for manipulating syntax, it is important enough to dictate our choice. This option is therefore disqualified as well. Together with not being able to use a '`quote`' object as discussed above, we are forced to use some form of a syntactic *descriptions* as quotations rather than proper quotations.

3.3.3 Using Standard Type Definitions

The naive approach to representing syntax, would be to imitate the implementation's term type as a system-level type. This is the approach taken by Aitken [3]. There are, however, some inherent problems with this:

1. First of all, it violates the main principle of exposing existing information — the implementation has some term type which is re-implemented at the

user level. This is in contrast to using a direct reflection which leads to the advantages that were described in Chapter 2.

2. This is manifested in leading to a re-implementation of term-related functionality, which requires intensive and non-trivial amounts of work. (Aitken's text [3] is close to 500 pages with around a 1/3 of that dedicated to representing terms, and it is still incomplete.)

3. Another consequence is that claiming that this representation is actually a reflection of Nuprl's syntax requires proving that both term types are equivalent. In Aitken's work this is not a problem since it implements the "paper definitions" of the syntax — resulting in reflecting this syntax, while it is possible that the implementation is different than both.

4. It requires a theory that can define such a type — we need to have tuples, lists, atoms etc. This might not be available in every setting, for example, working in a MetaPRL context that does not include ITT.

5. The size of a quoted term is exponential in its "quotedness" level, because a quotation of a quotation includes a representation of the constructors of the first representation. This is analogous to Scheme with no '`quote`' context, where quoting '`(+ 1 2)`' over and over leads to:

   - '`(list (quote +) 1 2)`'
   - '`(list (quote list) (list (quote quote) (quote +)) 1 2)`'
   - ...

6. A related problem to the previous one is that these representations are incomprehensible — the second-level quotation above contains enough "representation noise" to make the original term invisible.

The exponential representation can be improved using a different representation — as an encoding of a term, we can use an additional integer that specifies how quoted the term is. This means that to quote a quoted term all we need is to increase this number, but the resulting encoding would be harder to define and formalize, and term operations would become more complex: quoting a term twice could be done in two ways, either repeating the same process, or increasing the level for the second level. Another possible remedy would be using Nuprl's definition system — once we have quotation, we can have definitions that involve quotations, and quote these instead of re-quoting the original values. This would improve the term size problem, but not eliminate it, since terms that are quoted multiple times still need to be expanded during processing. The main point here is that it might be possible to overcome this problem in an ad-hoc way, but doing this might be inelegant and complex to formalize.

As for the problem of re-implementing internal functionality — it might seem that it could be solved by interfacing the object-level and the meta-level types.

This would take the form of being able to translate the object-level representation to internal term structures. The problem with this is that this translation must take place internally, making it already some form of exposure therefore changing the nature of the implementation, plus, the translation will not help in getting the facts about this internal functionality. The conclusion is that it might be possible given enough efforts, making it harder than approaching the problem with a direct reflection in mind.

Clearly, this approach does not look promising, an assumption that is supported by the efforts of Aitken.

3.3.4 Operator Shifting

A different kind of solution was first suggested by Aaron and Allen [1]. The idea is that for every possible operator o in the system, there is a matching operator o' that is used to create syntactic values that *mention* usages of the original operator. o' is called a *shifted*[3] o, and it is always a canonical value term. This makes it possible to use a form of quotation that is actually a *direct representation*, that is very convenient to use as well as providing quasi-quotation-like functionality.

Using shifted operators, it is important that every term will have a corresponding shifted version, including shifted terms. Shifted operators are syntactic — they consume and produce syntax representations. This is just a design sketch — there are two questions that need an answer for an actual implementation:

1. What is the relation between given syntactic values and the term whose representation is formed?

2. How does a usage of a shifted operator stand for a usage of the original one?

The first question determines the way terms are represented. One approach is using Nuprl definitions: no matter how terms are actually represented, a shifted operator has an implicit abstraction definition that unfolds to the actual representation of the unshifted term — independent of the way this representation is implemented, which makes this work even in the case of a recursive data type (but note that this does not help solving the problems mentioned above). The alternative approach would be making shifted operators be new primitive terms. Since our goal is to ultimately expose the internal structure, there is no way to define their meaning within the system and so we have to make shifted terms be *new primitives*.

The second question is about the structure of shifted operator usages, and will be discussed in Section 3.4 below.

[3]Note that Aaron&Allen do not use this name.

3.3.5 Quasi-Quotations

As mentioned above, it is crucial to have some form of quasi-quotation when we want to manipulate syntax. In a world where we want to talk about terms, it is essential that we have the expressive power that allows us to conveniently mix descriptions and quotations. On one hand, as said above, not being able to use quotations will make for a system that verbose to the point of being incomprehensible by humans to use as well as requiring solutions for problems like the exponential representation. On the other hand, giving up on all descriptions will force arcane solutions involving operations like 'replace' to imitate descriptions.

Quasi-quotations provide the best of both worlds: using them, you can specify mostly-literal quotations and occasionally jump back up to a meaningful description.

3.3.6 Using Preprocessing

A good preprocessing facility can be used to cover up for a lack of quotations or quasi-quotations. This is demonstrated by the approach that is commonly taken by Scheme implementations: the primitive built-in syntactic facilities that most implementations has two main features:

- First, there is an evaluation-stopper special form, '`quote`', responsible for creating a context where code stands for its syntax rather than its normal meaning.
- Second, there is the usual exposure of syntax objects as lists, symbols, and other datums.

Building on these two, '`quasiquote`' is implemented as a macro that translates its body into a mixture of list-operations and quoted code pieces. For example, the expression[4]:

```
‘(foo ‘(f (,g) ,,x) ,y)
```

will usually macro-expand to:

```
(list 'foo (list 'quasiquote (list 'f '(,g) (list 'unquote x))) y).
```

Note how the expansion of this expression uses '`quote`' to get symbol values as well as literal quoted subterms. This is just for optimizations: in fact, using a quasi-quotation macro, we can do with only list constructors and a primitive facility to quote identifiers as symbol values; eliminating the need for the proper quotation usage of '`quote`'. In this example, '`'(,g)`' would be replaced with '`(list (list 'unquote 'g))`'. Notice that this resolves the exponential user-code problem, but still an exponential blowup happens during evaluation, since the expansion has the full (exponentially big) form. This is the approach that is used by CamlP4 [25]: a certain input token causes the parser to read an expression and emit constructor code that generates the expression syntax rather than the expression itself.

[4] 'x' stands for '`(quasiquote` x`)`' and '`,`x' for '`(unquote` x`)`'

Indeed, this approach can be used with any representation, even flat strings or Gödel numbers: all we need is a preprocessing facility that is powerful enough to both encode and decode syntax representations. The only news here is shifting such code from being a required piece of functionality (a 'parse' function and its reverse), into the user-interface so the system never deals with unparsed data. The preprocessor (and a matching display mechanism) abstract over the actual representation, which makes its choice independent of the actual user-visible mechanism. We should, however, consider the mental price of a complex implementation which would result from inappropriate flat representations, the robustness price that will be noticeable if the resulting environment is bug-prone, and the convenience (and resource savings) of an efficient representation.

In Nuprl, the editor fills the role of such preprocessing via display forms and the structure editor. Regardless of the actual syntax representation we choose to work with, we should use these facilities to at least imitate quasi-quotation, allowing the desired intermix of quotations and descriptions. Theoretically, even the recursive data type approach could be made usable with such display forms, except that it would be hard to implement and still suffer from the discussed problems.

A generic parser for a language needs minor modifications for using this approach: say that we have a '`parse`' function that reads in text and returns some abstract syntax object. We wrap it in a new '`qparse`' function that

- scans the input for quotation marks and invokes itself recursively on their contents increasing a quotation-level flag,
- returns the same result as '`parse`' if the quotation level is zero, or converts the parsed result into a parsed form of the constructors that generate the syntax if the level is more than zero (repeating for each level).

For example, when reading '`a << b+1 >> c`', '`qparse`' will invoke itself recursively to read the '`b`', and instead of returning a '`b+1`' identifier syntax object, it will return the syntax for generating such an object (*e.g.*, the syntax of a '`addition(id("b"),num("1"))`' expression).

A more important point in all this is that if we implement such preprocessing over some existing syntax, it is extremely easy to come up with an additional markers that switch the quoted level down instead of up, thereby giving us instant quasi-quotations functionality. Modifying the above input to '`a << $b$+1 >> c`' will now return '`addition(b,num("1"))`'. This implementation technique is used extensively in CamlP4. Furthermore, if there is a syntactic representation for the operations for shifting quoted levels, then we get the ability to use arbitrary quotation levels.

The simplicity of using quasi-quotes and matching unquotes comes from their natural view as templates with holes to be filed. A very brief experience with these is enough to get convinced by their usefulness. It is therefore desirable to achieve such convenience when dealing with reflected Nuprl syntax.

```
(module opshift mzscheme
  (define-syntax opshift-app
    (syntax-rules ()
      [(_ 'op x ...) (list 'op x ...)]
      [(_ x ...)     (#%app x ...)]))
  (provide (all-from-except mzscheme #%app)
           (rename opshift-app #%app)))
```

Figure 3.2: Operator Shifting in PLT-Scheme

3.3.7 Operator Shifting and Quasi-Quotations

The technique of operator shifting is particularly suitable for mixing quotations and descriptions in a way that is almost as easy as using Scheme's '`quasiquote`' special form. This is quite obvious given the way a quotation is constructed using operator shifting — the term must be recursively traversed and all operators shifted; if some operator in the resulting structure is left unshifted, it keeps its usual meaning. A quick Scheme simulation can clarify this: Figure 3.2 contains a simple PLT-Scheme module that extends application expressions with 'shifted' function symbols denoted by a quoted identifier. As an example of using this, the expression

```
('+ 1 ('* '2 'x))
```

evaluates to (represents)

```
(+ 1 (* 2 x)).
```

Note that shifted operators are syntactic constructors: they operate on, and return syntactic values. Also note that this code works for multiple quotation levels:

```
(''+ 1 (''* ''2 ''x))
```

evaluates to

```
('+ 1 ('* '2 'x)).
```

Now, if the '`x`' symbol was not quoted, it would have the same meaning as a variable reference. This holds for operators as well,

```
> (define (add1 exp) ('+ exp '1))
> ('* '2 (add1 ('- 'x 'y)))
(* 2 (+ (- x y) 1))
```

which is equivalent to the following conventional Scheme code:

```
> (define (add1 exp) `(+ ,exp 1))
> `(* 2 ,(add1 `(- x y)))
(* 2 (+ (- x y) 1))
```

It is also evident that this representation is efficient: quoting '`(+ 1 2)`' three times requires a constant price for each element:

```
(’’’+ ’’’1 ’’’2),
```

which is not as good as a using contexts:

```
’’’(+ 1 2),
```

but much better than an exponential blowup such as:

```
(list ’list
      (list ’quote ’list)
      (list ’list (list ’quote ’quote) (list ’quote ’+))
      (list ’list (list ’quote ’quote) (list ’quote ’1))
      (list ’list (list ’quote ’quote) (list ’quote ’2)))
```

Finally, note that the shifted operator representation is naturally visualized by using a quotedness-level indicator color. This will be useful for implementing a user interface that is as provides an easier interface than Scheme's quasi-quote mechanism when used in an interactive structure editor. For example, compare this example from a proof that will be presented in Chapter 6 (using underline instead of colors):

`subx(subx(t;r); e) = subx(subx(t;e); subx(r;e))`

and the way that this would look if we had used a Scheme-like quasiquotation contexts:

`subx(q(subx(uq(t); uq(r))); e) = q(subx(uq(subx(t;e)); uq(subx(r;e))))`

3.4 Operator Shifting Options: Dealing with Bindings

Of all options that were discussed so far, operator shifting seems like the most convenient solution, and therefore it is the chosen implementation approach. For this, a new parameter family of natural number values is defined and used for quoted terms: ‘`rquote`’[5]. Plain terms are considered as implicitly holding a zero ‘`rquote`’ value; when a term is shifted, we simply add an ‘`rquote`’ parameter with a value of one, and when a shifted term is shifted again, the parameter value is increased. This plays well with Nuprl: as described in the beginning of this chapter, a term's meaning is determined by its signature (see Section 3.2.1), which contains its parameter values — so a shifted term has a meaning that is distinct from its unshifted form, and since these terms are not primitive terms and not user-defined abstractions, they are considered canonical value terms by default. Furthermore, we benefit from other term related functionality that uses term signatures to identify terms. For example, it is possible to specify display forms for shifted terms of a specific operator id and make them display in some unique way. The fact that we use a new ‘`rquote`’ parameter family ensures that shifted operators are genuinely new — it was impossible to have such terms before

[5] ‘`quote`’ was already used in Nuprl, so we choose ‘`rquote`’ as a mnemonic for ‘reflective quote’.

this extension, so there is no chance of having accidental terms that now have a different meaning[6].

Before we discuss the additional implementation details, there is still a question that requires an answer: *what shall we do with bindings of shifted operators?*

At first, it seems reasonable to materialize binding names. This is similar to Scheme quotations, where a quoted expression has concrete binding names: '`(lambda (x) x)`' has the same semantics as '`(lambda (y) y)`', but the corresponding quoted expressions are obviously different. This happens as a result of the representation of identifiers and quoted identifiers: the former are variables which are used only to connect binding positions and bound occurrences — the name '`x`' in '`(lambda (x) x)`' has no significance except for tying it to the (single) bound instance. Quoted identifiers are, in contrast, actual values, that can be used in any way. There are several indications of the very different nature of these two kinds of objects:

- First, in the current Scheme community, it is common to refer to 'identifiers' as the name for syntactic variables, and to 'symbols' as the plain values. It is also interesting to note that the Lisp community has a different approach to global bindings and to macros[7], and 'symbols' are the common term for both cases.

- The identifier—symbol difference is encouraged by hygienic macro implementations. These systems typically deal with bindings by hanging more information (a 'color') on symbols that are bindings, and use it do distinguish bindings rather than rely on their names.

- In fact, the whole issue of hygienic macros can be viewed as trying to compromise the concrete simple symbol view and the identifier view. Instead of '`defmacro`', we get '`syntax-rules`' which specifies rewrite rules, and identifiers need not be reified as concrete symbols. It can be claimed that Scheme's reflection is lacking in that bindings are not really reflected, and facilities that support hygienic macro compensate for this deficiency.

In Scheme, an expression's binding structure is determined implicitly by the special form that is used: primitive forms like '`lambda`' and '`let`' have their bindings at known locations, and user defined macros have binding positions that are determined by their expansion into primitive forms. With Nuprl terms, however, we face a different situation: terms have an explicit, uniform structure for bindings, making the binding structure part of a term's 'binding shape'. This has a major effect on the way bindings are treated when shifting operators: concrete,

[6] In contrast to ad-hoc mechanisms like using a new name for shifted operator ids.

[7] Use symbol properties for global bindings; use plain '`defmacro`' instead of hygienic macros, and use un-interned symbols in macros: symbols that have a unique identity regardless of their actual name.

Scheme-style quoting can only be achieved at a high price — the binding structure of a term and its shifted representation must be different. This is the same situation we have in Scheme (quoted terms have different binding structure: they never bind), but in Nuprl this difference is much more dramatic since bindings have a dedicated place in the syntax. This is demonstrated in the following section.

3.4.1 Concrete Bindings

Let us consider using concrete binding names. To quote a term that contains bound variables such as 'λ`x.x+1`', or in a more verbose form:

`lambda(x.add(x;1))`,

we need to shift all operators by adding an '`rquote`' parameter (marked as '`q`'):

`lambda{1:q}(x.add{1:q}(x;1))`.

This is not complete — bound variable occurrence and the integer are also terms that need to be shifted:

`lambda{1:q}(x.add{1:q}(variable{1:q,x:v}();natural_number{1:q,1:n}()))`,

but to reduce verbosity, we will use the same shorthand notation for integers and variables, but underline them when they are shifted:

`lambda{1:q}(x.add{1:q}(`$\underline{\texttt{x}}$`;`$\underline{\texttt{1}}$`))`,

we can further use this notation to show that any operator id is shifted:

$\underline{\texttt{lambda}}$`(x.`$\underline{\texttt{add}}$`(`$\underline{\texttt{x}}$`;`$\underline{\texttt{1}}$`))`,

and since punctuations are part of the syntax of terms, we underline additional characters that are associated with shifted operators:

$\underline{\texttt{lambda(x.add(x;1))}}$.

Now we get to the binding: we want concrete access to the variable name, some value that we can compare with the quoted variable name '$\underline{\texttt{x}}$' which is already a concrete value (it is a canonical form). The binding position cannot be shifted since as described above, it is a simple string rather than a term. This means that we need another ("binding") instance of '$\underline{\texttt{x}}$' in the quoted term, and that the existing binding should be removed, for example:

$\underline{\texttt{lambda(x; add(x;1))}}$.

The shape of the term has changed now — it no longer has the binding structure that it had before it was shifted. Specifically, we lost the binding structure information, for example, the result is ambiguous since we will get the same result if we shift the term '`lambda(x; add(x;1))`'.

To make this a proper term representation, we therefore need to add the binding structure information back into shifted operators. For example, we can decide that a shifted operator has all of its bindings in concrete form in the first places, and that there is another parameter value that holds the arity of the original term as a list of non-negative integers. The general rule would be that shifting

$$opid\{v{:}f,\ldots_k\}(x,\ldots_m.\,t;\ldots_n)$$

yields

$$opid\{\texttt{1:q,}[m,\ldots_n]\texttt{:a,}v{:}f,\ldots_k\}(\underline{x};\ldots_m;\ldots_n;t;\ldots_n).$$

This change in structure is not trivial, it requires a translation step that can

be considered as some form of encoding, it requires its reverse as some form of parsing, and quoted terms are very different than the terms they stand for. In short, this is simply using terms as a data structure that encode terms in an indirect manner. In addition to the necessary complexity that is involved, when we get to the semantic relation between terms and their quoted forms, we will need yet additional functionality to deal with bindings — namely alpha-equivalence related functionality. If we choose this strategy, then we might as well make it even more explicit and put the arity information as the first subterm, we can use a generic '`quote`' operator id and have the string and the parameter/value list as additional subterms, and this effectively throws us back to the new data-type approach which is undesirable.

This flaw stems from the fact that it diverges from the principle of direct reflection: instead of exposing the term structure, we encode it in a way that requires extra machinery.

3.4.2 Abstract Bindings

To rectify this situation, we consider leaving binders as they are when a term is shifted. Going back to the above sample term, 'λ`x.x + 1`', we shift it in the same way until we reach

`lambda(x.add(x;1))`.

If we now decide to keep the '`x`' binding position as is, then we need to change its bound instance back to a plain variable instance — otherwise we get a binding name that does not bind any instance, which clashes with Nuprl's semantics. Variable terms in Nuprl look like other terms, but they have a special status as they are linked to a binder that forms their scope. For example, when a binding name is renamed, all included variable subterms that are bound occurrences of the same name are renamed accordingly. In addition, we said that Nuprl assumes that subterms can always be substituted by equal subterms, but this does not hold for variable terms, since free variables are meaningless in the system.

The term that we now have,

`lambda(x.add(x;1))`,

is therefore broken: there is no relation between the binding position and the bound variable. We might try to fix this by treating bound variables as such regardless of being shifted or not, but then shifted bound variables are no longer concrete: it should not be possible to use the name of such a variable since renaming should not change the semantics of terms.

The approach we use to address these problems is to simply leave both binding positions and bound variables *unchanged* when a term is shifted. Shifting the original sample term will therefore yield:

`lambda(x.add(x;1))`.

The semantic implications of this representation are extremely important: we get an *abstract* representation of syntax instead of a fully opaque concrete structure. This leads to the semantics being defined using substitution functions as the main

building blocks, which lead to the core of this work. This will be thoroughly discussed in the following chapter.

The most obvious advantage of this shifting scheme is that it is as close to direct reflection of terms as we can get, making it follow the general principles laid out in the previous chapter for favoring exposure of internal functionality over re-implementation. Terms are quoted as shifted that have the exact same binding structure, and more importantly, bindings of shifted terms are the same bindings used for unshifted terms. The *only* operation that is needed when shifting and unshifting terms is to set the '`rquote`' parameter value. We immediately benefit from Nuprl's extensive term functionality — most notably is substitution: since shifted terms have normal bindings, Nuprl will avoid name captures, use consistent renaming based on the original names etc.

In fact, it seems that the only disadvantage of this approach is losing the ability to handle concrete syntax, in particular, we cannot use it to play with explicit names. As we shall see, this is not a problem, and we do not lose any expressiveness.

3.5 Technical Details

The design decision that is made at this point is to indeed use operator shifting with abstract bindings for term representations. We now briefly describe the actual implementation of this representation and some of the related technical points. Each of the following subsections corresponds to a piece of the implementation.

First of all, as said above, using direct reflection has its expected benefits. Substantial functionality that is already part of the system will deal with shifted operators in the same way, and since it is the same code that deals with unshifted terms, we are automatically guaranteed that the behavior on all levels is fully consistent. The implementation therefore contains only the minimal code that is required to establish shifted terms, and related user interface functionality. For example, we do not need to implement anything that deals with bindings, or to construct and access terms.

3.5.1 The '`rquote`' Parameter

The implementation consists of code written in Lisp, ML, and a Nuprl theory file. The Lisp part is responsible for two things which must be implemented at the system level: the addition of the '`rquote`' parameter and extensions to the display mechanism. A very small part of the code implements the '`rquote`' parameter: one definition adds the parameter type as a container for non-negative integer values, and a few more are used as new ML primitives to deal with constructing and destructing '`rquote`' values.

There is an implementation assumption that is made at this level and propagates to the ML code as well: an term is shifted only if it its first parameter is an

'`rquote`' which determines its quotedness; and unshifted terms do not have such a parameter. This is required so that unshifted terms have the same signatures as they used to have, allowing existing code to work unmodified.

3.5.2 Display Enhancements

We have already seen that using quotes and quasi-quotes expressions in Scheme is not a necessary feature, yet it makes dealing with syntax an order of magnitude easier. This is mainly due to making it as close to proper quotations as possible: a quoted expression contains a pictorial version of the expression it represents. Our quotation representation is simple to specify and work with, but it is also important to make it as accessible as possible for users. For this, we wish to visualize quoted terms in a way that *strongly relates* them to the term they stand for — make them display the *same* as the original terms but using a different color to indicate the quotedness of terms.

The first step in achieving this is extending the windowing user interface with the capability to show text in different colors. Nuprl uses a simple (and quite old) xlib-based interface: the foundation of this interface is the ability to display arrays of characters as interactive windows, working with terms goes through several processing steps which eventually generate such an array that Nuprl users work with via the windowing system. To enable colored display of terms, this fundamental part of the system is extended so it is possible to color text. This is a rather technical extension: character arrays contain 8-bit values that are ASCII with additional mathematical symbols, and our extension makes it use higher bits to specify an index into a vector of predefined colors.

This extension can be further used in the future to account for additional font properties (*e.g.*, super/sub-script, boldness, etc).

3.5.3 Display Forms

The most complicated part of the Lisp part of our implementation deals with Nuprl's term rendering engine. We extend it, as sketched above, to visualize shifted terms just like their unshifted versions but using a different color. Considerable effort has been invested on what seems like a technicality, but to reiterate the motivation: a good interface that involves an almost proper quotation (one that has a direct pictorial relation to the quoted syntax) is essential for effective and fruitful interactions.

To display a term, Nuprl follows these steps:

1. find a matching display form object (usually from the library) based on the term's signature, use a default display form if none found;

2. combine the term with the display form in a recursive process that matches content holes in the display form with subparts of the term (subterms, bindings, parameters, or operator id), the result of this is called a *display tree*;

3. process the display tree (destructively) according to layout elements of the display form, inserting newlines and indentations so that the term fits the current window width, resulting in a *text tree* which still has a connection to the original term and the selected display form,

4. convert the result to a rectangular string array that is finally visualized in the interaction window.

Note that terms are the *only* objects that are displayed, since Nuprl objects are all terms — the same generic process is used to display rules, definitions, sequents, and even user-interface elements like buttons.

Every step in this process is modified in some way to implement visualization of shifted terms:

1. when searching for a display form object for a shifted term, the extended system will first do the same search as for all terms and if no display form is found — if this search succeeds then we have a shifted term that has its own display form defined so it is used, otherwise, repeat the search without the '`rquote`' parameter (which always succeeds since there is always a default display form that can be used) and modify the resulting display form with quotedness-level mark objects around text that belongs to the display form and parameter slots[8];

2. the code that combines the display form with the term into a display tree is not really modified, but since it uses the results of the previous step, the term that gets used might be different than the original term (it might not have an '`rquote`' parameter);

3. then, turning the display tree to a text tree is modified: the display tree reflects the term structure that is visible on the tree — specifically, it has a pointer to a term object which is the term that is now being displayed — the change is that the quotedness level of the corresponding term is taken into account and the output is now a text tree that holds character codes ('ichars') that have the character index in the lower 8 bits and higher bits hold the quotedness level of the responsible term;

4. finally, the back end part that visualizes character arrays is modified to use different colors to indicate quotedness, as described above.

There are a few additional modifications that are needed, for example, the mechanism that matches nested terms to determine whether parentheses are necessary is modified to ignore quotedness levels.

The bottom line effect of all this is that quoted terms are displayed exactly as their normal versions but in a different colors according to their quotedness level

[8] A display form consists of pieces of text and layout directives, as well as slots for values that are found in the term: parameters, bindings, and subterms.

(unless a specific display form is defined for the quoted terms). But there are yet more issues to consider.

Finite Number of Distinct Colors

The first problem is that there are a finite number of colors to encode quotedness level. This number can be arbitrarily high (by simply adding colors to the defined color vector in the modified xlib interface), but this is limited to what is easily visible: we do not want the difference between terms at different quotedness levels to be indicated by a subtle change like a deeper shade of blue!

The solution is therefore to implement an alternative indication system for quoted text: we use a '<n:...>' wrapper around text that originate from an n-th level quoted term. If the quotedness level of a term exceeds the number of colors in the color vector, then this alternative is used to indicate quotedness instead of color shifting. This looks a little too verbose to be read conveniently off the screen, but there are plenty of colors to use (12) before the system resorts to this solution — such quotedness levels are unlikely to be needed, surely not for terms that are used interactively.

Visualization for Printouts

Printouts present another problem: Nuprl is able to produce printed mathematical text via LaTeX, and the result is often used in papers, books, and presentations — using colors for printouts will therefore not work. The solution is to implement yet another visualization scheme: when a display tree is converted to a text tree, the normal behavior is to use bits higher than the first 8 bits to indicate a color index. When we render terms for printouts, we switch to a mode where a color index is translated into LaTeX macros that use underlines to indicate the index. This is the same notation that is adopted throughout this text.

Customization

The default behavior is determined by a Lisp parameter. This parameter indicates the current rendering mode that is used for shifted terms. It is one of the following symbols:

'normal': default behavior as described above.

'verbose': using textual markers (this is the strategy that is used when the quotedness level is too high).

'print': used to output the LaTeX macros as described above when rendering terms for printouts.

'nil': standard Nuprl behavior, no special treatment for shifted terms.

The default '`normal`' value can be changed to always get some of the other behaviors.

Additional Usages

A pleasant side-effect of the display extensions is that colors can be used for other purposes. The color marker display commands are exposed as elements that can be used in display forms to put colors around certain pieces of terms. For example, the reflection theory file defines display forms for 'hypertext' and 'hyperterm' elements which contain some text and some term respectively, and are highlighted using color. Obviously, one has to be careful with such usage so it is not confused for quoted terms. This can be easily solved, but was not found to be necessary so far. Such usage is described in Section 6.1.

Coloring Bindings

Finally, dealing with bindings is, again, a point that needs careful consideration.

Display forms have 'slots' for various elements that come from the term that is rendered: parameters, bindings, and subterms. Subterms are visualized using their own display forms (which interact with the display form of the parent term, *e.g.*, for precedence and for parentheses), but parameters and bindings are rendered by the display form of the term they belong to.

So far, we have decided that:

- when a term is recursively shifted by quotation, binding positions stay the same, and subterms that are bound variables are therefore kept unshifted;
- the operator id part of a visualized term, and other pieces of text that are in its selected display form, use different colors to indicate different levels of quotedness of the term.

Parameters are rendered by the term's display form for a good reason: we view them as attributes that belong to the syntax of their term. Indicating parameter quotedness is no different — a quoted term represents some other term, including its parameters, so it is natural to use the same color for a term's name and for its parameters. For example, to indicate a shifted '`foo{1:n}`' term we write '`foo{1:n}`', and also for shorthand notation: a shifted '`1`' is printed as '`1`'[9].

We need to decide how to visualize binding positions[10]. Generally, binding positions are similar to parameters — they are associated with their surrounding term rather than the subterm; one technical reason for this is that the fundamental building block that is used in Nuprl is a term rather than a bound-term. This is not an arbitrary decision: the motivation is that terms can bind names, and not

[9] In Nuprl, this shorthand notation is implemented via a display form that omits the operator id.

[10] A question that would not arise if we were using a concrete syntax.

that some subterms happen to have bound variables. For example, in 'λ`x.x + 1`', we consider the 'λ' to be a binding term — it will not make sense if it has more or less subterms, or if it has more or less bindings[11].

If we follow this similarity and render binders using the same approach that is used for parameters, then a quoted 'λ`(x.x + 1)`' should be visualized as '$\underline{\lambda}$`(x.x`$\underline{+}$`1)`'. This is, however, highly misleading: the only meaningful property of bindings is the way they connect binding positions with bound instances — and '$\underline{\lambda}$`(x.x`$\underline{+}$`1)`' it. This property is fundamental enough that it is absurd to expect it to make sense to humans.

A possible solution is to visualize this term as: '$\underline{\lambda}$`(x.x`$\underline{+}$`1)`', which seems like a reasonable rendering since quoting a term results in a uniform shift in color. Implementing this visualization is a little tricky: before we convert a display tree into a text tree, we scan the tree structure recursively keeping an environment that matches bound names to their quotedness levels, and when we reach a variable subterm we change it to have the quotedness level of the term that binds it. However, this leads to ambiguity, confusing variables with quoted variables, for example, evaluating the following term[12]:

`apply(`λ`(var.`$\underline{\lambda}$`(x.x`$\underline{+}$`var)`$\underline{)}$`; `$\underline{\texttt{x}}$`)`

yields

$\underline{\lambda}$`(x.x`$\underline{+}$`x)`,

but the two '$\underline{\texttt{x}}$'s are different: only the second is actually shifted. If we now see '$\underline{\lambda}$`(x.x)`', we cannot know whether the subterm is a bound instance or a quoted variable term.

The basic problem stems from a *conflict between two desired properties*:

1. Quoted terms should look the just like their unquoted versions — no additional characters.

2. When a term gets quoted we want to see a uniform color shift — no different colors.

If we choose to color terms according to their actual quotedness, we get the first property, but we break the second one since bound instances are not shifted. If we choose the scheme that was just described, we get the two properties but at the price of ambiguity which disqualified it — but we can fix things, by breaking the first property. The idea is that quoted variable terms are not too common[13], yet they are *the only source of confusion*. So we change the way quoted variable terms are displayed by prefixing a quote character to their rendering, making the above computation be rendered as:

`apply(`λ`(var.`$\underline{\lambda}$`(x.x`$\underline{+}$`var)`$\underline{)}$`; `$\underline{\texttt{'x}}$`)` $\quad\rightarrow\quad$ $\underline{\lambda}$`(x.x`$\underline{+}$`'x)`

[11] This is why the arity list of a term is part of its signature.

[12] Notice how this demonstrates the quasi-quote-like feature of operator shifting.

[13] In Chapter 4 we will see that quoted variables behave just like Scheme symbols.

Indeed, the first property is violated since there are extra characters now, but this might be a feature as quoted variable terms are *very different* from real ones.

An alternative visualization strategy is simple: instead of making bound instances match binders by making them appear quoted, we go the other way and make the binders appear unquoted. With this approach, binders are not treated the same as parameters, instead, they are always shown the same way as unshifted terms. This approach violates the second feature: quoting any term that has bindings will not change all colors uniformly, but it is justified in that it shows exactly what the underlying terms are.

This discussion has led to implementing several strategies, and let a user-settable flag choose which one is used. This flag could be one of the following symbols, demonstrated using the quotation of '`λ(x.add(x;y))`':

'`simpler`': color terms according to their true quotedness, variables are treated like parameters;
Example: '`λ(x.add(x;y))`'

'`simple`': similar to '`simpler`' but show variables like unshifted parts;
Example: '`λ(x.add(x;y))`'

'`uniform`': color both binding positions and bound occurrences according to the quotedness of the binding term.
Example: '`λ(x.add(x;y))`'

The '`uniform`' option was used by default, but when the semantic account was formalized, it became clear that quoting terms using real (abstract) bindings has a major effect on the semantics, and that the '`simple`' visualization is a better choice. Eventually, it was kept as the only colorization approach.

3.5.4 Quote-Related Functionality

Miscellaneous quote-related functionality is implemented in ML. The only functionality that is needed at the Lisp level is the addition of the '`rquote`' parameter type, and higher level operations are better left for ML. The design follows an assumption that was already mentioned: an unshifted term has no '`rquote`' parameter, and a shifted term has a single one which comes first.

The main part of this layer implements quote and unquote functions ('`rquote_term`' and '`runquote_term` respectively). These would be written as a straightforward recursive scan of input terms, but we need to avoid shifting variable terms that are bound by shifted operators. Fortunately, Nuprl's standard ML library contains a '`sweep_up_map_with_bvars`' function which maps a function recursively, collecting and passing the list of variable names that are currently bound.

Notice that at the ML level, Nuprl users deal with concrete syntax, unlike the abstract semantic level. The chosen quotation representation uses abstract variable, but implementing a quote and an unquote function is done at the underlying

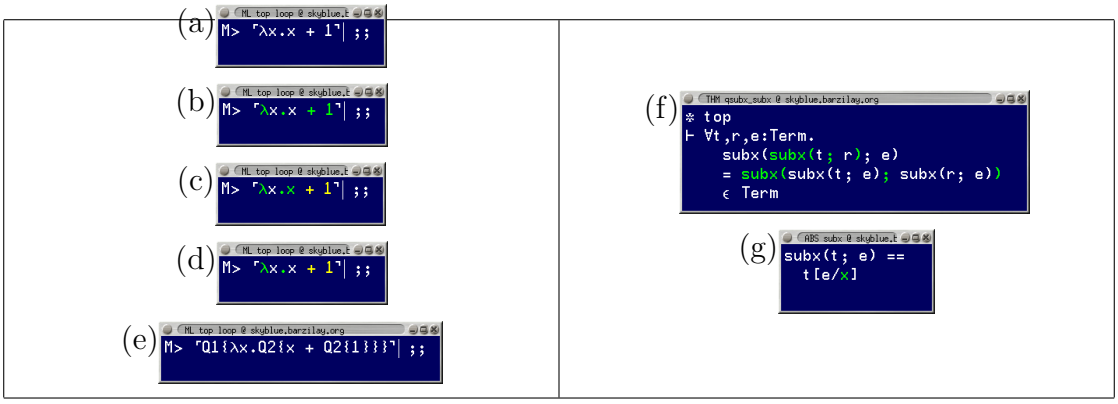


Figure 3.3: Interacting with quoted terms using colors

concrete level. As we shall later see, this is similar to other systems that use higher-order abstract syntax.

Due to the fact that these functions are implemented at the concrete meta-level, they are not intended for users. Nuprl has a complex keyboard macro system that is used to configure editor input. This macro system can call ML functions to operate on the currently selected term. We add a few such macros that use the '`rquote`' and '`runquote`' functions to shift the current term up or down.

The ML implementation has additional quoted-term-related functions which are used for evaluation hooks and for supporting a few rules. They are described later.

3.5.5 Reflection Theory

Finally, the last part of our implementation is a Nuprl theory. This is where the implementation is tied to its intended semantics. The fact that we are using direct reflection is evident in that quoted terms are handled by the same mechanisms that deal with unquoted terms — but, as discussed in the previous chapter, Nuprl's term implementation is operational in its nature, while we need more for Nuprl to be able to use term representations in Nuprl's user level. Missing from the implementation are the formal properties of terms and term functionality. This "semantical glue" is included in the reflection theory file, defining rules (axioms) and derived theorems that formalize term properties.

The reflection library is described and included in Appendix B.1. It is described in more details in Chapter 6.

3.6 Usage Sample

To demonstrate the way users interact with the system, we present a few screen-shots in Figure 3.3.

(a) This is a picture of Nuprl ML 'toploop' window, after the term 'λ`x.x + 1`'

was entered.

(b) Selecting the whole term, and using the key combination that quotes it results in this state. Note that, as discussed above, the binding position is not colored as quoted even though it comes from a shifted term, and the corresponding bound occurrence is not shifted quoted.

(c) Selecting the addition subexpression and quoting it results in this term. The problem is that the term that was quoted contained a free occurrence of ‘x’, so it was quoted — breaking the original binding structure. For such situations, we have also added key bindings that can shift single term without the recursive scan that quotes it with its subterms.

(d) This shows the previous term after it was fixed. Note that the meaning of this term is not trivial: it is the quotation of a λ-function that returns an addition expression with its argument as the first subterm and ‘1’ as the second. In Scheme, using concrete syntax, this would be written as:

```
‘(lambda (x) ‘(+ ,x 1))
```

It should be clear now that using colors is as simple as using quasi-quotes in Scheme. This is in contrast to the constructor approach of:

```
(list 'lambda (list 'x)
  (list 'list (list 'quote '+) 'x (list 'quote '1)))
```

or, using some possible ML-like syntax:

```
lambda x. make_addition(x, make_number(1))
make_lambda("x",
            make_application
              (make_var("make_addition"),
               make_var("x"),
               make_application("make_number",
                                make_number(1))))
```

which is barely comprehensible.

(e) The same expression is displayed in the ‘verbose’ mode. This is a little confusing (*e.g.*, the ‘x’ occurrence is unshifted since it doesn't have a direct ‘Q’ wrapper), and with some work could be displayed better, perhaps in a more quasi-quote-like manner. This is not a practical problem though: 12

levels of quotations are represented using colors, and it is highly unlikely that more than four levels of quotation will ever be needed. (*e.g.*, stating the Gödel theorem requires three levels, and understanding it completely requires four.)

(f,g) These two pictures show actual usages of quoted terms as syntax constructors, (f) is an initial specification of a proof goal, and (g) is an abstraction definition that uses term substitution[14] with a quoted variable — as we shall see, this is similar to using a Scheme symbol.

[14]One of the operations that are computed via hooks in the ML part of our implementation. It will is discussed later.

Chapter 4
Semantics of Shifted Terms

In the previous chapter we have chosen a representation for syntax, we demonstrated its practicality for users and its computational advantages, and outlined implementation issues. It is obvious that our representation makes it possible to quote all terms by a recursive process of adding an '`rquote`' parameter to a term and its subterms except for bound variable subterms, but we still need to clarify the semantics of shifted terms.

This chapter will discuss the problematic issues that are involved, and present a valid semantical account. The presentation is generic: it does not depend directly on the specifics of Nuprl terms or on its type theory. We only need a simple term structure (operator ids and bound subterms), and few assumptions and facts to hold (*e.g.*, the ones that are mentioned in Section 4.4.1).

4.1 Brief Review

Operator-denoting operators are called *shifted* operators: if an operator x denotes the operator y, then x is called a shifted y, and will be typeset as $\underline{y}$. For example, '$a \underline{+} b$' denotes '$c+d$' if a denotes c and b denotes d. The plus operator, '+', denotes a function that takes two integers and returns an integer, and its shifted version, '$\underline{+}$', denotes a function that takes two terms and returns a term. In the previous chapter, we have discussed the advantages of keeping the same binding structure when shifting a term — now the problem is what do we do with such operators: for example, '`∀x.P(x)`' is an operator that denotes a function taking a boolean or propositional function and returning a boolean or a proposition (its syntactic form is, of course, binding). (We use a '∀' operator to avoid the confusion that would result if a 'λ' term was used; note that Nuprl's actual '∀' operator takes another argument for the type, *e.g.*, '`∀x:ℕ.P(x)`'.)

The obvious choice for the semantics of the shifted version, if we were to use a concrete representation, would be a function, '$\underline{\texttt{all(x;P)}}$' that takes two expressions as input values: one for the bound name and one for the body, and constructs the *concrete* '∀' term. We use an abstract syntax representation however, and the semantics of our syntax representation requires using functions, making it a *higher-order abstract syntax* [57]. Going in this direction, we get the usual benefits of this approach over concrete syntax (or alternatives like de-Bruijn indexes), such as specified by Pitts and Gabbay [59], as well as the benefits of direct reflection that were discussed in Chapter 2. In particular, it allows us to retain the same binding structure as the operator being denoted. For example, the single input argument for '$\underline{\forall}$' has the same binding as '∀': it takes in a term-valued *function* as an argument (a single-variable bound subterm).

We begin by asking what is the semantics of '$\underline{\forall}$'? The semantics of a *concrete* shifted '∀' is the trivial one given above, but the semantics for our abstract '$\underline{\forall}$' is

more subtle.

Note: we use an overline to indicate sequences and a way to index their elements. For example, $\overline{x} : \mathsf{Var}^n$ means that $\overline{x}$ is a sequence of n Vars, and that x_i is the ith element of this list; that is, x is a function from $i : 1 \ldots \mathrm{len}(\overline{x})$ to the ith element of $\overline{x}$. This is essentially the same as the ellipsis notation that was introduced in Section 3.1, only this is somewhat more convenient to use in bigger formulas, in particular, we use the overline notation with simple sequences rather than the general ellipsis notation capability of specifying more complex rewrite rules. Also, we use only indexed functions to stand for these sequences.

4.2 Semantics of Shifted Operators

Since '$\underline{\forall}$' is a binding operator, it takes a function as an argument. Our basic requirement is that '`F(t)`' be the result of the 'All-Instantiation' rule applied to '$\underline{\forall}$`x.F(x)`' and '`t`'. This means that '`F`' *needs to be a substitution function.* So the semantics we adopt for '$\underline{\forall}$`x.F(x)`' is that it denotes the '$\forall$' formula whose predicate part is '`F(`u`)`' and whose binder is u for some u — almost.

But which u? As usual we can avoid this question by using a higher-order abstract syntax, and say that what is denoted is actually the α-equivalence class of all such formulas where some appropriate u could be found. From this point forth, we use 'Term' to refer to these α-equivalence classes rather than the concrete terms.

Before going on to the technical parts, lets consider how we might reason about this in the reflective logic. The first intuition is that proving that something is a Term depends only on having a quoted operator '$\underline{\mathsf{opid}}$' and on its subparts in a simple compositional way:

$\vdash \underline{\mathsf{opid}}(\overline{v}.b;\ldots) \in \mathsf{Term}$
if $\overline{v} : \mathsf{Term} \vdash b \in \mathsf{Term}$
$\vdots$

This seems fine, but it fails with bound variables. For example, the following can be proved:

$\vdash \underline{\lambda}\mathsf{x}.\mathsf{if}\ \mathsf{x} = \underline{0}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{2} \in \mathsf{Term}$
because $\mathsf{x} : \mathsf{Term} \vdash \mathsf{if}\ \mathsf{x} = \underline{0}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{2} \in \mathsf{Term}$

The premise line is trivial, but the conclusion is false, because the quoted $\underline{\lambda}$-term contains a function which is not a substitution function — it is not a "template" function. In other words, there is no literally quoted term that this value stands for; indeed, there is no way to unquote this term.

When inspecting this term, we can compare it to similar but valid terms to see what went wrong with the suggested rule:

1. $\underline{\lambda}\mathsf{x}.\mathsf{if}\ \mathsf{x} = \underline{0}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{2}$
2. $\underline{\lambda}\mathsf{x}.\mathsf{if}\ \mathsf{x} = \underline{0}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{1}$

3. $\lambda \mathtt{x}.\mathsf{if}\ \mathtt{x} = \underline{0}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{2}$

The two $\underline{\lambda}$-terms are fine, because they're built from substitution functions, and the last one is a simple $\mathsf{Term} \to \mathsf{Term}$ function. The difference between these terms and the previous one indicates what is wrong with the above rule: the bound variable should *not be used as a value*. It is a binding that should only be used in template holes, as there is no real value that this variable is ever bound to that can be used. In the valid examples, the first one does exactly that: it does not *use* the bound value except for sticking it in its place. The second one 'almost' uses the value, but since the two branches are identical it is possible to avoid evaluating the test term; therefore it can be evaluated without using it. The last one is not a Term but a function on Terms, so it can use that value as usual.

The conclusion is that a bound variable can be used only as an argument of a quoted term constructor. In other words, it can serve only as a value that is "computationally inert", much like universe expressions in Allen's thesis [4]. This is also similar to variables that are bound by Scheme's '`syntax-rules`' [44] — they are template variables that can be used in syntactic structures only to build new structures[1]. When put in this light, it seems that any attempt to get this property in a proof fails. The lesson from this is: variables bound by quoted operators do not behave like normal bindings in the sense that they do not provide any values that are usable at the normal Nuprl level — a fact that is also true in regards to universe expressions.

4.3 Term Definition

We take CTerms as *concrete terms*: the type of objects intended to be ordinary syntax objects with binding operators, and concrete names (ignoring parameters for simplicity). A more precise definition is given later, in Section 4.5. To define the Term type, we also need to introduce a predicate, '`is_subst`', which is used to distinguish proper substitution functions. This predicate is defined in Section 4.6, and it has specific rules which are introduced in Section 4.6.1.

As said above, we define Terms using CTerms and α-equality:

$$\mathsf{Term} \equiv \mathsf{CTerm}//\alpha$$

Terms are constructed by shifted operators, which have the semantics of functions that create Terms from Term substitution functions. For example, '$\underline{\lambda}$' is a function that takes a single-argument proper substitution function and constructs a Term

[1] For example, in a '`syntax-rules`' template of '`((foo x) (bar x))`', the identifier '`x`' is just a place holder that can be used to stick a value in a template; it is not possible to inspect its value. This is why '`syntax-rules`' is considered part of a high-level macro language, strictly less powerful than a lower-level base functionality that implements it.

value:

$$\underline{\lambda} : \{f : \mathsf{Term} \to \mathsf{Term} \mid \mathtt{is\_subst}_1(f)\} \to \mathsf{Term}$$

using '$\mathtt{is\_subst}_1$', which is a version of '$\mathtt{is\_subst}$' that works with one argument functions. In general, '$\mathtt{is\_subst}_n$' is a predicate over $\mathsf{Term}^n \to \mathsf{Term}$. To simplify things, we drop the n when the context makes it clear.

CVar is a subset of CTerm, which contains only atomic variable terms. Correspondingly, $\mathsf{Var} = \{\{x\} \mid x \in \mathsf{CVar}\}$, therefore $\mathsf{Var} \subseteq \mathsf{Term}$. This is a set of singletons since variables are α-equivalent *only* to themselves. Two assumptions that will be used in the following text are that we have an infinite supply of distinct variables in CVar (and therefore in Var) and that there is at least one closed CTerm we can use.

4.4 Operations, Assumptions, and Facts

These are the operations that will be needed in the following text:

$\cdot\uparrow$ taking the α-equivalence class of an object.

$\cdot\downarrow$ choosing an element of an α-equivalence class. This is some function, such as one that chooses the first available variable names using lexicographic order.

$\cdot[\cdot/\cdot]$ standard capture-avoiding substitution on CTerms. It can be used to substitute multiple variables in one shot, provided that the number of supplied terms matches the number of variables, which are all distinct.

$\cdot[\![\cdot/\cdot]\!]$ substitution for Terms, which is defined using the above operations as: $b[\![\overline{x}/\overline{v}]\!] = b\downarrow\,[\overline{x}\downarrow/\overline{v}\downarrow]\uparrow$.

$\mathrm{newcvar}(\cdot)$ returns a new CVar, i.e., $\mathrm{newcvar}(t)$ is neither free nor bound in $t \in \mathsf{CTerm}$.

$\mathrm{newvar}(\cdot)$ is similar to $\mathrm{newcvar}(\cdot)$ but for Terms, defined as:

$$\mathrm{newvar}(x) = \mathrm{newcvar}(x\downarrow)\uparrow\ .$$

$\mathrm{newcvar}_n(\cdot)$ returns n new CVars, defined as:

$$\begin{aligned} \mathrm{newcvar}_1(x) &= \mathrm{newcvar}(x), \\ \mathrm{newcvar}_{n+1}(x) &= (\text{let } v = \mathrm{newcvar}_n(x) \text{ in } v, \mathrm{newcvar}(v, x)). \end{aligned}$$

$\mathrm{newvar}_n(\cdot)$ returns n new Vars, defined using $\mathrm{newvar}(\cdot)$ in the same way as $\mathrm{newcvar}_n(\cdot)$.

We use versions of these operations that are generalized to any lists and tuples of input arguments in an obvious way. The newcvar$(\cdot)$ and newvar$(\cdot)$ operations are further extended to functions by plugging in some closed dummy term argument (that we name '0') and using the result:

$$\forall f : \mathsf{Term}^n \to \mathsf{Term}.\ \mathrm{newvar}_m(f) = \mathrm{newvar}_m(f(0^n))$$

Below we will often justify things of the form $a\downarrow = b\downarrow$, by mentioning lemmas of the form $a =_\alpha b$, without emphasizing the required transitions. Note that we use the overline notation (introduced in page 59) extensively.

4.4.1 Important Assumptions and Facts

We now list several assumptions and derived facts about substitution — the assumptions are not argued for, but we think that it is clear they are all true for any reasonable definition of substitution (one that respects the usual term binding structure); specifically, we trust Nuprl's substitution to satisfy these assumptions. This allows us to take substitution as given and avoid getting into the specifics of an implementation. The assumptions and facts that are introduced here will be used in the following text[2].

$\star 1$ $\forall x : \mathsf{Term}.\ x = x\Downarrow$

$\star 2$ $\forall x : \mathsf{CTerm}.\ x =_\alpha x\Downarrow$
This fact is mostly used when nested in a bigger term, see $\star 4$ below.

$\star 3$ $\forall x : \mathsf{CVar}.\ x = x\Downarrow$
because $x \in \mathsf{CVar} \ \Rightarrow\ x\Downarrow = \{x\}\downarrow = x$.

$\star 4$ $\forall \overline{x_1}, \overline{x_2} : \mathsf{CTerm}^n, \overline{v} : \mathsf{CVar}^n,\ b : \mathsf{CTerm}.\ \overline{x_1} =_\alpha \overline{x_2} \ \Rightarrow\ b[\overline{x_1}/\overline{v}] =_\alpha b[\overline{x_2}/\overline{v}]$
Note that using this fact, $\star 2$ can be used in a subterm of an α-equality, since: $\forall t, x : \mathsf{CTerm}.\ t =_\alpha t[x\Downarrow/x]$

$\star 5$ $\forall b_1, b_2 : \mathsf{CTerm},\ \overline{t} : \mathsf{CTerm}^n,\ \overline{v} : \mathsf{CVar}^n.\ b_1 =_\alpha b_2 \ \Rightarrow\ b_1[\overline{t}/\overline{v}] =_\alpha b_2[\overline{t}/\overline{v}]$
Note that $\overline{v}$ is the same on both sides (free variables in the body are not changed).

$\star 6$ $\forall t : \mathsf{CTerm},\ \overline{x_1} : \mathsf{CTerm}^{n_1},\ \overline{x_2} : \mathsf{CTerm}^{n_2},\ \overline{v_1}, \overline{u} : \mathsf{CVar}^{n_1},\ \overline{v_2} : \mathsf{CVar}^{n_2}.$
the sequence $\overline{v_1}, \overline{v_2}$ are distinct & $\overline{u}$ are distinct, not free in $t, \overline{x_2}$
$\Rightarrow\ t[\overline{x_1}, \overline{x_2}/\overline{v_1}, \overline{v_2}] =_\alpha t[\overline{u}, \overline{x_2}/\overline{v_1}, \overline{v_2}][\overline{x_1}/\overline{u}]$
This is simple to verify:

$$\begin{aligned}
&t[\overline{u}, \overline{x_2}/\overline{v_1}, \overline{v_2}][\overline{x_1}/\overline{u}] \\
&\qquad =_\alpha t[\overline{u}[\overline{x_1}/\overline{u}], \overline{x_2}[\overline{x_1}/\overline{u}]/\overline{v_1}, \overline{v_2}] && \text{(any of } \overline{u} \text{ do not occur free in } t\text{)} \\
&\qquad =_\alpha t[\overline{x_1}, \overline{x_2}[\overline{x_1}/\overline{u}]/\overline{v_1}, \overline{v_2}] && \text{(}\overline{u} \text{ are distinct)} \\
&\qquad =_\alpha t[\overline{x_1}, \overline{x_2}/\overline{v_1}, \overline{v_2}] && \text{(any of } \overline{u} \text{ do not appear in } \overline{x_2}\text{)}
\end{aligned}$$

[2] We use "$\star N$" to refer to fact N.

Note that it is easy to show that such a $\overline{u}$ exists by choosing it as:
let $\overline{u} = \text{newcvar}_{n_1}(t, \overline{x_2}, \ldots)$

$\star 7$ $\forall t : \mathsf{Term},\ \overline{x_1} : \mathsf{Term}^{n_1},\ \overline{x_2} : \mathsf{Term}^{n_2},\ \overline{v_1}, \overline{u} : \mathsf{Var}^{n_1},\ \overline{v_2} : \mathsf{Var}^{n_2}$.
the sequence $\overline{v_1}, \overline{v_2}$ are distinct & $\overline{u}$ are distinct, not free in $t, \overline{x_2}$
$\Rightarrow\ t[\![\overline{x_1}, \overline{x_2}/\overline{v_1}, \overline{v_2}]\!] = t[\![\overline{u}, \overline{x_2}/\overline{v_1}, \overline{v_2}]\!][\![\overline{x_1}/\overline{u}]\!]$
Again, verifying this is simple: from $\star 6$ we know that

$$t\lfloor\ [\overline{u}\rfloor, \overline{x_2}\lfloor/\overline{v_1}\rfloor, \overline{v_2}\lfloor][\overline{x_1}\lfloor/\overline{u}\rfloor]\rceil = t\lfloor\ [\overline{x_1}\rfloor, \overline{x_2}\lfloor/\overline{v_1}\rfloor, \overline{v_2}\lfloor]\rceil,$$

so:

$$\begin{aligned}
&t[\![\overline{u}, \overline{x_2}/\overline{v_1}, \overline{v_2}]\!][\![\overline{x_1}/\overline{u}]\!] \\
&\qquad = t\lfloor\ [\overline{u}\rfloor, \overline{x_2}\lfloor/\overline{v_1}\rfloor, \overline{v_2}\lfloor]\rceil\!\rfloor\ [\overline{x_1}\lfloor/\overline{u}\rfloor]\rceil && (\cdot[\![\cdot/\cdot]\!] \text{ definition}) \\
&\qquad = t\lfloor\ [\overline{u}\rfloor, \overline{x_2}\lfloor/\overline{v_1}\rfloor, \overline{v_2}\lfloor][\overline{x_1}\lfloor/\overline{u}\rfloor]\rceil && (\star 2, \star 5) \\
&\qquad = t\lfloor\ [\overline{x_1}\rfloor, \overline{x_2}\lfloor/\overline{v_1}\rfloor, \overline{v_2}\lfloor]\rceil && (\text{by the use of } \star 6 \text{ above}) \\
&\qquad = t[\![\overline{x_1}, \overline{x_2}/\overline{v_1}, \overline{v_2}]\!] && (\cdot[\![\cdot/\cdot]\!] \text{ definition})
\end{aligned}$$

A similar note holds here: it is easy to show that such a $\overline{u}$ exists if it is chosen as:

$$\text{let } \overline{u} = \text{newcvar}_{n_1}(t\lfloor, \overline{x_2}\lfloor, \ldots\lfloor)\rceil = \text{newvar}_{n_1}(t, \overline{x_2}, \ldots)$$

$\star 8$ $\forall c : \mathsf{CTerm},\ \overline{v}, \overline{u} : \mathsf{CVar}^n,\ \overline{s}, \overline{t} : \mathsf{CTerm}^n$.
$\overline{u}$ are not free in c except for $\overline{v}\ \Rightarrow\ c[\overline{s}/\overline{v}][\overline{t}/\overline{u}] =_\alpha c[\overline{s}[\overline{t}/\overline{u}]/\overline{v}]$
Note that the $\overline{v}$ exception is usually not needed.

$\star 9$ $\forall c : \mathsf{Term},\ \overline{v}, \overline{u} : \mathsf{Var}^n,\ \overline{s}, \overline{t} : \mathsf{Term}^n$.
$\overline{u}$ are not free in c except for $\overline{v}\ \Rightarrow\ c[\![\overline{s}/\overline{v}]\!][\![\overline{t}/\overline{u}]\!] = c[\![\overline{s}[\![\overline{t}/\overline{u}]\!]/\overline{v}]\!]$
This is easily shown by $\star 2$ and the definition of $\cdot[\![\cdot/\cdot]\!]$, using the previous fact.

A general intuition that arises from these facts and others, is that Term values are indeed isomorphic to CTerms: as long as there are no "dirty" concrete tricks played by *using* names of bound variables, facts that hold for CTerms will have corresponding versions for Terms.

4.5 Definitions of Shifted Operators

In the general case, a *shifted* operator id, '<u>opid</u>', is defined as a function that takes in some substitution functions (verified by '`is_subst`') of some arities, and returns a Term value. This is done in the obvious way: each of the substitution functions is used to plug in new variables; then the results, together with the chosen variables, are all packaged into a CTerm; and finally, the α-equivalence class of this result produces the resulting Term. The actual representation is not too important: we use the direct reflection mechanism that was introduced in Chapter 3, so the representation that is used here is only for the formal account. We need some

new data types for this, and the simple pairs and lists approach could be used, for example:

$$\underline{\lambda}(f) = \langle \text{`}\lambda\text{'}, [\langle [\text{newvar}(f)\rfloor], f(\text{newvar}(f))\lfloor \rangle]\rangle\rceil$$

but this gets too complex for our purpose, *e.g.*, making analysis hard since we need to distinguish pairs that stand for a bound term, a term, or a pair of terms. If were using the new data type approach for the actual implementation, we would probably use the same representation for the formal account, as was done by Aitken [3].

Instead, we use some new types that are not fully specified, requiring only a few properties and some abstract operations. Instead of a full specification of these types, we use them as given and specify only the corresponding constructors. As said above, we use indexed functions instead of tuples or lists for sequences, since they make later parts of this account somewhat easier. The new types that we now need are:

- OpId will be used for term name labels;
- $\mathsf{BndCTerm}_a$ is a *bound* CTerm (where $a : \mathbb{N}$) — packaging a CTerm with a distinct CVars.

BndCTerms are created with a `mkBndCTerm` constructor[3]:

$$\texttt{mkBndCTerm} \quad \in \quad a : \mathbb{N} \to (1 \ldots a \to \mathsf{CVar}) \to \mathsf{CTerm} \to \mathsf{BndCTerm}_a$$

An alternate syntax for `mkBndCTerm` uses our sequence notation, and can be more natural when a is known:

$$\texttt{mkBndCTerm}(\overline{x}, t) \quad \text{stands for} \quad \texttt{mkBndCTerm}(\text{len}(\overline{x}), (\lambda i.\ x_i), t)$$

which is used as if the function accepts a tuple rather than a size and an indexed function:

$$\texttt{mkBndCTerm} \quad \in \quad \mathsf{CVar}^a \to \mathsf{CTerm} \to \mathsf{BndCTerm}_a$$

CTerms are created with `mkCTerm`:

$$\begin{aligned} \texttt{mkCTerm} \quad \in \quad & \mathsf{OpId} \to n : \mathbb{N} \\ & \to a : (1 \ldots n \to \mathbb{N}) \\ & \to (i : 1 \ldots n \to \mathsf{BndCTerm}_{a_i}) \\ & \to \mathsf{CTerm} \end{aligned}$$

Note that we assume OpId exists and values of this type are accessible in some (unspecified) way. Again, an alternate syntax for this uses sequence notation that can be more natural when n, a are known is:

$$\texttt{mkCTerm}(o,\ [\texttt{mkBndCTerm}(\overline{x_1}, t_1), \ldots, \texttt{mkBndCTerm}(\overline{x_n}, t_n)])$$

[3]We use the notation $x : A \to B_x$ to denote functions on A such that $\forall x : A.\ f(x) \in B_x$, a type which is more conventionally denoted by $\Pi x : A.\ B_x$.

which stands for

$$\texttt{mkCTerm}(o,\ n,\ (\lambda i.\ \text{len}(\overline{x_i})),\ (\lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}, t_i)))$$

The next thing we need is a type which is the subset of $\mathsf{Term}^n \rightarrow \mathsf{Term}$ functions that are substitution functions (using the '`is_subst`' predicate, defined below):

$$\mathsf{SubstFunc}_n = \{f : \mathsf{Term}^n \rightarrow \mathsf{Term} \mid \texttt{is\_subst}_n(f)\}$$

Now we have reached the point where we can finally define a `mkTerm` constructor for Terms, one that is based on `mkCTerm`. Its type is:

$$\begin{aligned}\texttt{mkTerm} \quad \in \quad & \mathsf{OpId} \rightarrow n : \mathbb{N} \\ & \rightarrow a : (1 \ldots n \rightarrow \mathbb{N}) \\ & \rightarrow (i : 1 \ldots n \rightarrow \mathsf{SubstFunc}_{a_i}) \\ & \rightarrow \mathsf{Term}\end{aligned}$$

and it is defined as:

$$\begin{aligned}&\texttt{mkTerm}(o, n, a, f) = \\ &\quad \texttt{mkCTerm}(o, n, a, \lambda i.\ \text{let } \overline{x} = \text{newvar}_{a_i}(f_i) \text{ in } \texttt{mkBndCTerm}(\overline{x}\lfloor, f_i(\overline{x})\lfloor))\rceil\end{aligned}$$

The alternate syntax for this is:

$$\texttt{mkTerm}(o, [\langle a_1, f_1\rangle, \ldots, \langle a_n, f_n\rangle])$$

which stands for

$$\texttt{mkTerm}(o, n, \lambda i.\ a_i, \lambda i.\ f_i) = \texttt{mkTerm}(o, n, a, f)$$

Shifted operators can now be defined as Term constructors which use `mkTerm` with some fixed operator name and arity list. For example, '$\underline{\lambda}$' and '$\underline{\Sigma}$' are defined, using the alternate syntax notation, as[4]:

$$\underline{\lambda}(f) = \texttt{mkTerm}(\text{`}\lambda\text{'}, [\langle 1, f\rangle]), \quad \underline{\Sigma}(f, g) = \texttt{mkTerm}(\text{`}\Sigma\text{'}, [\langle 0, f\rangle, \langle 1, g\rangle])$$

Note that since `mkTerm` is curried, a shifted operator is made by specifying the first three inputs: $\texttt{mkTerm}(o, n, a)$; this information corresponds to term signatures in Nuprl.

In addition to the assumptions and facts that were introduced in Section 4.4.1, we further assume the following:

$\star$10 We specify one way that substitution interacts with CTerms: for all i, k, if it is true that

$$\text{if } v_k \text{ is free in } t_i \text{ then none of } \overline{x_i} \text{ are free in either } r_k \text{ or } v_k$$

[4]These definitions use '$\underline{\lambda}$' and '$\underline{\Sigma}$' as functions, not as operator ids.

then[5],

$$\begin{aligned}&\texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}, t_i))[\overline{r}/\overline{v}]\\ &\qquad =_\alpha \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}, t_i[\overline{r}/\overline{v}]))\end{aligned}$$

To see why it is true using any reasonable definition of substitution, it is simpler to first see that a precondition that could be used is that none of $\overline{x_i}$ occur free in $\overline{r}, \overline{v}$; this is too restrictive for our future needs but the explanation is somewhat similar.

First of all, if v_k is not free in t_i, then there is no need for any restriction, since it does not have any effect on the result. Now, if it does appear in t_i, then it is enough to have two guarantees for the above to remain an α-equality: (a) if none of $\overline{x_i}$ are free in r_k then capture by $\overline{x_i}$ is impossible; (b) if v_k is not in $\overline{x_i}$, then none of the v_k will get "screened out" in the body.

- It seems that a fact similar to this assumption also holds for Terms — that if none of $\overline{x_i}$ occur free in $\overline{r}, \overline{v}, \overline{f_i(\underline{0}^{a_i})}$ then:

  $$\texttt{mkTerm}(o, n, a, \lambda i.\ f_i)[\![\overline{r}/\overline{v}]\!] = \texttt{mkTerm}(o, n, a, \lambda i.\ \lambda\overline{z}.\ f_i(\overline{z})[\![\overline{r}/\overline{v}]\!])$$

  However, it turns out that this fact is incorrect, but the concrete version is the only one we need.

⋆11 A simple fact about renaming bound variables:

$$\begin{aligned}\forall \overline{x_i}, \overline{z_i} : \mathsf{CVar}^n.\ &\overline{x_i} \text{ are distinct } \&\ \overline{z_i} \text{ are distinct } \&\ \overline{z_i} \text{ are not free in } b_i\\ &\Rightarrow\ \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}, b_i))\\ &\qquad =_\alpha \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{z_i}, b_i[\overline{z_i}/\overline{x_i}]))\end{aligned}$$

4.6 Defining '`is_subst`'

A function is a *substitution function* iff there exists an appropriate *substitution* that it is equivalent to. We describe this first using CTerms, since we know how substitutions work on them:

$$\texttt{is\_subst}_n(f) \ \equiv\ \exists b : \mathsf{CTerm}.\ \exists \overline{v} : \mathsf{CVar}^n.\ \forall \overline{t} : \mathsf{CTerm}^n.\ f(\overline{t}\restriction) = b[\overline{t}/\overline{v}]\restriction \qquad (4.1)$$

Note that f returns a Term which is an α-equivalence class, so the above is an equality that compares two such classes rather than an α-equality over concrete terms. This should be equivalent to directly using a Term argument for f:

$$\texttt{is\_subst}_n(f) \ \equiv\ \exists b : \mathsf{CTerm}.\ \exists \overline{v} : \mathsf{CVar}^n.\ \forall \overline{r} : \mathsf{Term}^n.\ f(\overline{r}) = b[\overline{r}\downarrow/\overline{v}]\restriction \qquad (4.2)$$

We show that $\forall b : \mathsf{CTerm},\ \forall \overline{v} : \mathsf{CTerm}^n$, the two sub-expressions are equivalent.

[5]Note that the α-equality is needed only because the substitution definition might introduce arbitrary renamings.

(4.1) ⇒ (4.2) Instantiate $\overline{t}$ with the chosen $\overline{r}\downarrow$:

$$\begin{aligned} f(\overline{r}) &= f(\overline{r}\Downarrow) && (\star 1)\\ &= b[\overline{r}\downarrow/\overline{v}]\uparrow && (4.1) \end{aligned}$$

(4.2) ⇒ (4.1) Instantiate $\overline{r}$ with $\overline{t}\uparrow$ and we get:

$$\begin{aligned} f(\overline{t}\uparrow) &= b[\overline{t}\Downarrow/\overline{v}]\uparrow && (4.2)\\ &= b[\overline{t}/\overline{v}]\uparrow && (\star 2 \star 4) \end{aligned}$$

We can now try to use a version that uses only Term types (no CTerm types), using α-terms substitution, $\cdot[\![\cdot/\cdot]\!]$:

$$\texttt{is\_subst}_n(f) \;\equiv\; \exists b_a : \mathsf{Term}.\, \exists \overline{v_a} : \mathsf{Var}^n.\, \forall \overline{t_a} : \mathsf{Term}^n.\, f(\overline{t_a}) = b_a[\![\overline{t_a}/\overline{v_a}]\!] \qquad (4.3)$$

and verify that this is indeed equivalent to the other two definitions:

(4.2) ⇒ (4.3) Let $b_a = b\uparrow$, $\overline{v_a} = \overline{v}\uparrow$, pick some $\overline{t_a}$, and instantiate $\overline{r}$ with it:

$$\begin{aligned} f(\overline{t_a}) &= b[\overline{t_a}\downarrow/\overline{v}]\uparrow && (4.2)\\ &= b\Downarrow [\overline{t_a}\downarrow/\overline{v}\Downarrow]\uparrow && (\star \text{ see below})\\ &= b\uparrow[\![\overline{t_a}/\overline{v}\uparrow]\!]\\ &= b_a[\![\overline{t_a}/\overline{v_a}]\!] \end{aligned}$$

($\star$) is true because of $\star 2$ (with b), $\star 3$ (with $\overline{v}$), and $\star 5$ (with $x_1, x_2, \overline{t}, \overline{v}$).

(4.3) ⇒ (4.2) Let $b = b_a\downarrow$, $\overline{v} = \overline{v_a}\downarrow$, pick some $\overline{r}$, and instantiate $\overline{t_a}$ with it:

$$\begin{aligned} f(\overline{r}) &= b_a[\![\overline{r}/\overline{v_a}]\!] && (4.3)\\ &= b_a\downarrow [\overline{r}\downarrow/\overline{v_a}\downarrow]\uparrow\\ &= b[\overline{r}\downarrow/\overline{v}]\uparrow \end{aligned}$$

4.6.1 The 'is_subst' Rule

Now that we have a reasonable definition of '`is_subst`', we define key rules that are used to prove an '`is_subst`' property, which is needed to verify that something is a proper Term. These rules are quite simple — there are only two cases:

- $H \vdash \texttt{is\_subst}(\overline{x}.\, x_i)$ where $1 \le i \le \text{len}(\overline{x})$
- $H \vdash \texttt{is\_subst}(\overline{x}.\, \underline{\text{opid}}(\overline{y}.\, b; ...))$ where $\underline{\text{opid}}$ is some shifted opid
  if $H \vdash \texttt{is\_subst}(\overline{x}, \overline{y}.\, b)$
  $\vdots$

These two rules are sufficient for any substitution function, which in turn is sufficient for proving the validity of any shifted term value that is a valid quotation. Proving $t \in \mathsf{Term}$ is achieved by showing `is_subst`$(.\ t)$. For example, to prove that '`foo(x.bar(1;y.x))`' is a Term:

- we begin with `is_subst`(. `foo(x.bar(1;y.x))`),
- '`foo`' is shifted so using the second rule we continue with `is_subst`(`x`. `bar(1;y.x)`),
- again, '`bar`' is shifted so we now have two subgoals for the two subterms:
  - `is_subst`(`x`. `1`): '`1`' is shifted and it has no subterms so it is trivially true using the second rule,
  - `is_subst`(`x`, `y`. `x`): the function in question is a simple projection function, so we're done using the first rule.

Of course, this is not a complete set of rules, since there are more cases where we have general Term expressions that are not constants but contain a mixture of shifted operators and descriptions. In such cases the H context is used with standard Nuprl rules.

4.6.2 Justifying the '`is_subst`' Rules

First Rule

The validity of the first rule amounts to this:

$$\forall n, i : \mathbb{N}^+.\ i \le n \ \Rightarrow\ \mathtt{is\_subst}_n(\pi^i_n)$$

which is easily verified. Choose distinct $\overline{v} = v_1, \ldots, v_n$ variables, and let $b = \pi^i_n(\overline{v}) = v_i$. Then, $\forall \overline{t} : \mathsf{Term}^n.\ \pi^i_n(\overline{t}) = v_i[\![\overline{t}/\overline{v}]\!]$ is true by the definition of π^i_n, of $\cdot[\![\cdot/\cdot]\!]$, and the distinctness of $\overline{v}$.

Second Rule

Our main result will be formulating and proving the validity of the second rule:

$H \vdash$ `is_subst`$(\overline{x}.$ opid$(\overline{y}.\ b; ...))$ where opid is some shifted opid
if $H \vdash$ `is_subst`$(\overline{x}, \overline{y}.\ b)$
$\vdots$

but this formulation requires some preparation. First, recall that the type of `mkTerm` is:

$$\mathsf{OpId} \to n : \mathbb{N} \to a : (1 \ldots n \to \mathbb{N}) \to (i : 1 \ldots n \to \mathsf{SubstFunc}_{a_i}) \to \mathsf{Term}$$

Note that, as said earlier, a shifted operator is the result of applying `mkTerm` on the first three arguments, since they define the operator symbol and the list of arities it expects. For example:

$$\underline{\lambda} = \texttt{mkTerm}(\text{`}\lambda\text{'}, 1, \langle 1 \rangle) \qquad \underline{\Sigma} = \texttt{mkTerm}(\text{`}\Sigma\text{'}, 2, \langle 0, 1 \rangle)$$

So, a shifted operator has the following type: for some given o, n, and a, it is a constructor that creates an o Term out of n substitution functions with the given a arities:

$$\texttt{mkTerm}(o, n, a) : \ (i : 1 \ldots n \to \mathsf{SubstFunc}_{a_i}) \to \mathsf{Term}$$

Remember that the current goal is to conclude that for some shifted operator, $\underline{\mathsf{opid}}$:

$$\texttt{is\_subst}(\overline{x}.\ \underline{\mathsf{opid}}(\overline{v_1}.\ b_1, \ldots, \overline{v_n}.\ b_n))$$
$$\text{if}$$
$$\texttt{is\_subst}(\overline{x}, \overline{v_1}.\ b_1)\ \&\ \ldots\ \&\ \texttt{is\_subst}(\overline{x}, \overline{v_n}.\ b_n)$$

We need to compose the $\underline{\mathsf{opid}}$ function with an object that will make the result a $\mathsf{Term}^k \to \mathsf{Term}$ function (consuming the $x_1, \ldots, x_k$ variables) which we then show is a substitution function. This means that the function that is composed with $\underline{\mathsf{opid}}$ should get a tuple of Term^k as input and return the vector of n substitution functions, built by consuming $\overline{x}$. In short, we package all the necessary information in F:

$$F : \mathsf{Term}^k \to (i : 1 \ldots n \to \mathsf{SubstFunc}_{a_i})$$

so we get the expected:

$$\texttt{mkTerm}(o, n, a) \circ F : \mathsf{Term}^k \to \mathsf{Term}$$

Now for the main result — the validity of the second rule may be formulated thus:

$$\begin{array}{l} \forall\ o : \mathsf{OpId},\ n : \mathbb{N},\ a : (1 \ldots n \to \mathbb{N}),\ k : \mathbb{N}, \\ \quad F : \mathsf{Term}^k \to (i : 1 \ldots n \to \mathsf{SubstFunc}_{a_i}). \\ \qquad \forall i : 1 \ldots n.\ \texttt{is\_subst}_{k+a_i}(\lambda_{(k)}\, ts, xs.\ F(ts)(i)(xs)) \\ \qquad\quad \Rightarrow\ \texttt{is\_subst}_k(\texttt{mkTerm}(o, n, a) \circ F) \end{array}$$

where $(\lambda_{(k)} x, y.\ B(x, y))(u_1, \ldots, u_{k+n}) \equiv B((u_1, \ldots, u_k), (u_{k+1}, \ldots, u_{k+n}))$.[6]

Proof. Assume o, n, a, k, and F are given as specified. We also assume that the constructed functions are substitution functions; therefore, for every $1 \le i \le n$ we get $c_i : \mathsf{Term}$, $\overline{u_i} : \mathsf{Var}^k$, $\overline{v_i} : \mathsf{Var}^{a_i}$ such that:

$$\forall r_1^1, \ldots, r_k^1, r_1^2, \ldots, r_{a_i}^2 : Term.\ F(r_1^1, \ldots, r_k^1)(i)(r_1^2, \ldots, r_{a_i}^2) = c_i[\![\overline{r^1}, \overline{r^2}/\overline{u_i}, \overline{v_i}]\!]$$

[6]Note that this special form of λ could be avoided if the fourth input type to `mkTerm` would take the terms first and then the index (instead of the $\mathsf{SubstFunc}_{a_i}$), but that would require a special composition operation instead.

- Let $\overline{t} : \mathsf{Term}^k$ be some k Terms,
- let $\overline{x_i} = \mathrm{newvar}_{a_i}(F(\overline{t})(i))$,
- and let $\overline{s} = \mathrm{newvar}_k(\overline{c}, \overline{x_1}, \ldots, \overline{x_n})$.

Now we can proceed: our goal due to the definition of '`is_subst`', is to derive an equality of the form

$$(\mathtt{mkTerm}(o, n, a) \circ F)(\overline{t}) = B[\![\overline{t}/\overline{X}]\!]$$

where, and this will be the tricky part, B and $\overline{X}$ are *independent* of the input, $\overline{t}$. So:

$$\begin{aligned}
&(\mathtt{mkTerm}(o, n, a) \circ F)(\overline{t}) \\
&\quad = \mathtt{mkTerm}(o, n, a, F(\overline{t})) \\
&\quad = \mathtt{mkCTerm}(o, n, a, \lambda i.\ \mathtt{mkBndCTerm}(\overline{x_i}\!\downarrow, F(\overline{t})(i)(\overline{x_i})\!\downarrow))\!\uparrow && (\mathtt{mkTerm}\text{ def.}) \\
&\quad = \mathtt{mkCTerm}(o, n, a, \lambda i.\ \mathtt{mkBndCTerm}(\overline{x_i}\!\downarrow, c_i[\![\overline{t}, \overline{x_i}/\overline{u_i}, \overline{v_i}]\!]\!\downarrow))\!\uparrow && (F\text{'s fact}) \\
&\quad = \mathtt{mkCTerm}(o, n, a, \lambda i.\ \mathtt{mkBndCTerm}(\overline{x_i}\!\downarrow, c_i[\![\overline{s}, \overline{x_i}/\overline{u_i}, \overline{v_i}]\!][\![\overline{t}/\overline{s}]\!]\!\downarrow))\!\uparrow && (\star 7) \\
&\quad = \mathtt{mkCTerm}(o, n, a, \lambda i.\ \mathtt{mkBndCTerm}(\overline{x_i}\!\downarrow, c_i[\![\overline{s}, \overline{x_i}/\overline{u_i}, \overline{v_i}]\!]\!\downarrow\ [\overline{t}\!\downarrow/\overline{s}\!\downarrow]\!\Downarrow))\!\uparrow && (\cdot[\![\cdot/\cdot]\!]\text{ def.}) \\
&\quad = \mathtt{mkCTerm}(o, n, a, \lambda i.\ \mathtt{mkBndCTerm}(\overline{x_i}\!\downarrow, c_i[\![\overline{s}, \overline{x_i}/\overline{u_i}, \overline{v_i}]\!]\!\downarrow\ [\overline{t}\!\downarrow/\overline{s}\!\downarrow]))\!\uparrow && (\star 2) \\
&\quad = \mathtt{mkCTerm}(o, n, a, \lambda i.\ \mathtt{mkBndCTerm}(\overline{x_i}\!\downarrow, c_i[\![\overline{s}, \overline{x_i}/\overline{u_i}, \overline{v_i}]\!]\!\downarrow))[\overline{t}\!\downarrow/\overline{s}\!\downarrow]\!\uparrow && (\star 10, \text{ see below}) \\
&\quad = \mathtt{mkCTerm}(o, n, a, \lambda i.\ \mathtt{mkBndCTerm}(\overline{x_i}\!\downarrow, c_i[\![\overline{s}, \overline{x_i}/\overline{u_i}, \overline{v_i}]\!]\!\downarrow))\!\Downarrow\ [\overline{t}\!\downarrow/\overline{s}\!\downarrow]\!\uparrow && (\star 2) \\
&\quad = \mathtt{mkCTerm}(o, n, a, \lambda i.\ \mathtt{mkBndCTerm}(\overline{x_i}\!\downarrow, c_i[\![\overline{s}, \overline{x_i}/\overline{u_i}, \overline{v_i}]\!]\!\downarrow))\!\uparrow[\![\overline{t}/\overline{s}]\!] && (\cdot[\![\cdot/\cdot]\!]\text{ def.})
\end{aligned}$$

In the above, making sure that $\star 10$ can be applied needs some care. Assume that for some j, l, the variable $s_j\!\downarrow$ is free in the lth body, which is $c_l[\![\overline{s}, \overline{x_l}/\overline{u_l}, \overline{v_l}]\!]\!\downarrow$. We need to make sure in this case that $\overline{x_l}\!\downarrow$ is not free in either $t_j\!\downarrow$ or $s_j\!\downarrow$. The latter is trivial by the choice of $\overline{s}$ (and holds for all indexes), but the former is not obvious. What we do know about $\overline{x_l}\!\downarrow$ is its definition:

$$\overline{x_l}\!\downarrow \;=\; \mathrm{newvar}_{a_l}(F(\overline{t})(l))\!\downarrow \;=\; \mathrm{newvar}_{a_l}(c_l[\![\overline{t}, 0^{a_l}/\overline{u_l}, \overline{v_l}]\!])\!\downarrow$$

but since $s_j\!\downarrow$ is free in $c_l[\![\overline{s}, \overline{x_l}/\overline{u_l}, \overline{v_l}]\!]\!\downarrow$, then $u_{l,j}\!\downarrow$ must appear in $c_l\!\downarrow$; therefore, the choice of $\overline{x_l}\!\downarrow$ above must pick variables that do not appear in $\overline{t_j}\!\downarrow$ so we're safe.

Going back to the main proof, the last term of the equality chain built so far was:

$$\mathtt{mkCTerm}(o, n, a, \lambda i.\ \mathtt{mkBndCTerm}(\overline{x_i}\!\downarrow, c_i[\![\overline{s}, \overline{x_i}/\overline{u_i}, \overline{v_i}]\!]\!\downarrow))\!\uparrow[\![\overline{t}/\overline{s}]\!]$$

which has the $B[\![\overline{t}/\overline{X}]\!]$ structure that we're looking for, but we're not finished because both the B and the $\overline{X}$ parts depend on $\overline{t}$ — $\overline{x_i}$ is defined in terms of $\overline{t}$, $\overline{s}$ is defined in terms of $\overline{x_i}$, and both B and $\overline{X}$ parts contain instances of $\overline{s}$ (and B actually contains $\overline{x_i}$ as well).

So we choose $\overline{t}$-independent values now: let $\overline{x'_i} = \mathrm{newvar}_{a_i}(c_i)$ and let $\overline{s'} = \mathrm{newvar}_k(\overline{c}, \overline{x'_1}, \ldots, \overline{x'_k})$, we also need to show that in the above, using $\overline{x'_i}, \overline{s'}$ instead

of $\overline{x_i}, \overline{s}$ is still the same value. In an attempt to simplify this we now choose n sets of variables $\overline{z_1} \in \mathsf{Term}^{a_1}, \ldots, \overline{z_n} \in \mathsf{Term}^{a_n}$, which are completely fresh: they do not appear in anything mentioned so far, including $\overline{t}$.

Now, back to our equality chain which left off at:

$$= \mathtt{mkCTerm}(o, n, a, \lambda i.\ \mathtt{mkBndCTerm}(\overline{x_i}\downarrow, c_i[\![\overline{s}, \overline{x_i}/\overline{u_i}, \overline{v_i}]\!]\downarrow))\uparrow[\![\overline{t}/\overline{s}]\!]$$

By $\star 11$:

$$= \mathtt{mkCTerm}(o, n, a, \lambda i.\ \mathtt{mkBndCTerm}(\overline{z_i}\downarrow, c_i[\![\overline{s}, \overline{z_i}/\overline{u_i}, \overline{v_i}]\!]\downarrow))\uparrow[\![\overline{t}/\overline{s}]\!]$$

Next, we use substitution to get $\overline{s'}$ inside — $\overline{s}$ are distinct, $\overline{s'}$ are distinct, and $\overline{s'}$ does not occur in $\overline{z_i}$:

$$= \mathtt{mkCTerm}(o, n, a, \lambda i.\ \mathtt{mkBndCTerm}(\overline{z_i}\downarrow, c_i[\![\overline{s'}[\![\overline{s}/\overline{s'}]\!], \overline{z_i}[\![\overline{s}/\overline{s'}]\!]/\overline{u_i}, \overline{v_i}]\!]\downarrow))\uparrow[\![\overline{t}/\overline{s}]\!]$$

Because $\overline{s'}$ does not occur free in c_i, this would be the expansion of the following substitution by $\star 9$:

$$= \mathtt{mkCTerm}(o, n, a, \lambda i.\ \mathtt{mkBndCTerm}(\overline{z_i}\downarrow, c_i[\![\overline{s'}, \overline{z_i}/\overline{u_i}, \overline{v_i}]\!][\![\overline{s}/\overline{s'}]\!]\downarrow))\uparrow[\![\overline{t}/\overline{s}]\!]$$

Combining $\cdot[\![\cdot/\cdot]\!]$ and $\star 2$ we get:

$$= \mathtt{mkCTerm}(o, n, a, \lambda i.\ \mathtt{mkBndCTerm}(\overline{z_i}\downarrow, c_i[\![\overline{s'}, \overline{z_i}/\overline{u_i}, \overline{v_i}]\!]\downarrow\ [\overline{s}\downarrow/\overline{s'}\downarrow]))\uparrow[\![\overline{t}/\overline{s}]\!]$$

$\overline{z_i}$ do not occur in either $\overline{s}$ or $\overline{s'}$ so we can use $\star 10$:

$$= \mathtt{mkCTerm}(o, n, a, \lambda i.\ \mathtt{mkBndCTerm}(\overline{z_i}\downarrow, c_i[\![\overline{s'}, \overline{z_i}/\overline{u_i}, \overline{v_i}]\!]\downarrow))[\overline{s}\downarrow/\overline{s'}\downarrow]\uparrow[\![\overline{t}/\overline{s}]\!]$$

Again, using $\cdot[\![\cdot/\cdot]\!]$ and $\star 2$:

$$= \mathtt{mkCTerm}(o, n, a, \lambda i.\ \mathtt{mkBndCTerm}(\overline{z_i}\downarrow, c_i[\![\overline{s'}, \overline{z_i}/\overline{u_i}, \overline{v_i}]\!]\downarrow))\uparrow[\![\overline{s}/\overline{s'}]\!][\![\overline{t}/\overline{s}]\!]$$

Now, $\overline{s}$ does not appear in the $\mathtt{mkCTerm}$ except possibly for $\overline{s'}$ (because we know it is not in c_i or $\overline{z_i}$), so using $\star 9$ we get:

$$= \mathtt{mkCTerm}(o, n, a, \lambda i.\ \mathtt{mkBndCTerm}(\overline{z_i}\downarrow, c_i[\![\overline{s'}, \overline{z_i}/\overline{u_i}, \overline{v_i}]\!]\downarrow))\uparrow[\![\overline{s}[\![\overline{t}/\overline{s}]\!]/\overline{s'}]\!]$$
$$= \mathtt{mkCTerm}(o, n, a, \lambda i.\ \mathtt{mkBndCTerm}(\overline{z_i}\downarrow, c_i[\![\overline{s'}, \overline{z_i}/\overline{u_i}, \overline{v_i}]\!]\downarrow))\uparrow[\![\overline{t}/\overline{s'}]\!]$$

Finally, using $\star 11$ we get:

$$= \mathtt{mkCTerm}(o, n, a, \lambda i.\ \mathtt{mkBndCTerm}(\overline{x'_i}\downarrow, c_i[\![\overline{s'}, \overline{x'_i}/\overline{u_i}, \overline{v_i}]\!]\downarrow))\uparrow[\![\overline{t}/\overline{s'}]\!]$$

Our final term has the desired $B[\![\overline{t}/\overline{X}]\!]$ form, and now the B and the $\overline{X}$ parts are independent of $\overline{t}$. This is because:

- $\overline{x'_i}$ depends only on c_i;
- $\overline{s'}$ depends only on $\overline{x'_i}$ and $\overline{c}$, and therefore only on $\overline{c}$;
- and $\overline{u_i}$ and $\overline{v_i}$, just like $\overline{c}$, were derived from the assumption that the inputs are substitution functions.

QED.

4.7 Relations to HOAS Work

We have shown the plausibility of basing logical reflection on a higher-order abstract syntax, where each syntactic operator is denoted directly by another operator. Our development of a quotation mechanism was independent of standard higher-order abstract syntax (HOAS) work, but the result closely parallels it. HOAS is a relatively new idea (late 80's) for a practical system — described by Pfenning and Elliott [57] (but suggested before that). In this section we present a survey of HOAS work, along with its relation to our work in Nuprl. A good introduction to HOAS was written by Hofmann [38], which is loosely used as the basis for some parts of this survey. Note that in recent years efforts in this area have been somewhat intensified, for example, the PoplMark Challenge [7] has been presented to the formal community and has been encouraging new work in the area. (In addition, this comes at a time where "language oriented[7]" tools are becoming more popular even in the commercial world.) This presentation is, therefore, likely to be somewhat dated.

The main goal of HOAS is representing formal languages that use bindings, in a way that relieves users from the hassle of name management — alpha-renaming being the major complication that needs constant care. This is achieved by a form of reflection: bindings of the object language are represented by bindings of the meta language, effectively exposing the binding management of the meta language implementation to the object language. This is achieved by using only the binding structure of functions, so syntax is represented using functions.

A standard concrete representation of terms using abstract syntax trees[8] would use the following type definition[9]:

$$\begin{aligned} \mathsf{Term} ::= \quad & \mathtt{var} : \mathsf{Var} \to \mathsf{Term} \\ | \quad & \mathtt{app} : \mathsf{Term} \times \mathsf{Term} \to \mathsf{Term} \\ | \quad & \mathtt{lam} : \mathsf{Var} \times \mathsf{Term} \to \mathsf{Term} \end{aligned}$$

Comparing this to the higher-order version:

$$\begin{aligned} \mathsf{Term} ::= \quad & \mathtt{app} : \mathsf{Term} \times \mathsf{Term} \to \mathsf{Term} \\ | \quad & \mathtt{lam} : (\mathsf{Term} \to \mathsf{Term}) \to \mathsf{Term} \end{aligned}$$

demonstrates the reflected usage of the meta-language bindings — a function is represented by a structure that itself holds a function, and there is no explicit

[7] There is no common term for this methodology yet.

[8] Abstract syntax trees, or AST's, is a common term for a representation of syntax using recursive data structures, unrelated to 'abstraction' as a term used for functions. See also the glossary entry.

[9] In this notation, typewriter fonts are used for constructor functions, for example, $\mathtt{lam} : \mathsf{Var} \times \mathsf{Term} \to \mathsf{Term}$ means that `lam` gets a variable and a Term and returns a Term. The '$\to$' arrow is the standard function type constructor.

variable name anywhere. For example, this is how the standard Church numeral "1" is represented:

$$1 = \lambda f x.\ f x \quad \text{is represented by:} \quad \texttt{lam}(\lambda f.\ \texttt{lam}(\lambda x.\ \texttt{app}(f, x)))$$

First, note that there are three different 'lambda' operators: the 'λ' on the left is part of the input syntax, the '`lam`' is the HOAS Term constructor mentioned above, and this has an argument which is a *meta-level* 'λ' which denotes a real function. Also note that the variables written in the object language are represented by meta-level variables in the HOAS representation: the same names are used for convenience but there are no concrete names in the representation — we could just as well write

$$1 = \lambda f x.\ f x \quad \text{is represented by:} \quad \texttt{lam}(\lambda a.\ \texttt{lam}(\lambda b.\ \texttt{app}(a, b))),$$

there is no '`var`' case since there is no need for one.

There is an additional benefit for using a HOAS representation: binding structures are represented as functions, so to substitute a term for a bound variable, all we need to do is apply the function that represent the binding structure. For example, beta reduction is specified succinctly as:

$$\texttt{app}(\texttt{lam}(f), t) \quad \text{reduces to:} \quad f(t)$$

In fact, this is an obvious usage given the way we have presented our Nuprl quotations — these are *substitution* functions precisely because they represent substitutions. Our semantic explanation is one that treats the property of representing substitutions at the same importance level as representing bindings, in contrast to standard HOAS work that seems to treat it merely as a convenient by-product.

The HOAS representation is implemented by a simple translation function that is used to quote input terms:

$$\begin{aligned} \ulcorner \lambda x.\ e \urcorner &\rightarrow \texttt{lam}(\lambda x.\ \ulcorner e \urcorner) \\ \ulcorner e_1 e_2 \urcorner &\rightarrow \texttt{app}(\ulcorner e_1 \urcorner, \ulcorner e_2 \urcorner) \\ \ulcorner x \urcorner &\rightarrow x \end{aligned}$$

This translation function is part of the implementation's interface code: it works with concrete syntax as its input and output.

Figure 4.1 demonstrates a naive Nuprl implementation of an '`aterm`' using the built-in concrete '`term`' type. This code is purely demonstrational, but a representation that is similar in its nature (and type) could be used to make Nuprl use these '`aterm`'s as the fundamental data type. This particular implementation suffers from relying on the concrete '`term`' type, but it is enough to demonstrate the principles and the problems that are involved in such a representation. For example note that, as expected, variable *names* are non-existent in the representation. Also note that '`term_to_aterm`' and its helper function, '`bterm_to_substfunc`' serve as the translation functions which work on the concrete syntactic level. More serious problems that such higher-order abstract representations suffer from are discussed below.

```
absrectype aterm =
  (tok # parm list) # ((int # (aterm list -> aterm)) list)
with make_aterm opsig substfuncs = abs_aterm (opsig,substfuncs)
and destruct_aterm aterm = rep_aterm aterm
;;

letrec new_vars v others n =
   if n>0 then
      let new = new_var v others in
         new . (new_vars v (new . others) (n-1))
   else []
;;

let zero_aterm =
  make_aterm (op_of_term (mk_integer_term 0)) [] ;;

letrec bterm_to_substfunc (vars, subterm) =
   length vars,
   (\ts .
      term_to_aterm (fo_subst (zip vars (map aterm_to_term ts))
                              subterm))
and substfunc_to_bterm (arity, func) =
   let bogus = aterm_to_term
                (func (build_list [arity,zero_aterm])) in
   let vars = new_vars `x' (free_vars bogus) arity
   in
      vars,
      (aterm_to_term
        (func (map (term_to_aterm o mk_variable_term)
                   vars)))
and term_to_aterm term =
   let opsig, bterms = destruct_term term in
      make_aterm opsig (map bterm_to_substfunc bterms)
and aterm_to_term aterm =
   let opsig, substfuncs = destruct_aterm aterm in
      make_term opsig (map substfunc_to_bterm substfuncs)
;;
```

Figure 4.1: A naive HOAS implementation using Nuprl's '`term`' type

4.7.1 HOAS Problems

As useful as HOAS is for managing bindings, it comes at a heavy price. There are two major problems: exotic terms and induction.

Exotic terms result from the fact that the functions used in the representation cannot be just any random $\mathsf{Term} \to \mathsf{Term}$ function, they must not *use* their input argument for anything except for syntactic constructors. For example, we can define an '`is_app`' function that returns a Church boolean depending on its input being an application construct:

$$\begin{aligned} \texttt{is\_app} &: \mathsf{Term} \to \mathsf{Term} \\ \texttt{is\_app}(\texttt{app}(t, r)) &= \lambda x.\ \lambda y.\ x \\ \texttt{is\_app}(\texttt{lam}(f)) &= \lambda x.\ \lambda y.\ y \end{aligned}$$

Since `is_app` has the correct type, we can use it with the HOAS `lam` constructor, `lam`(`is_app`), and the result is a closed Term value according to the HOAS Term definition. However, this construction does not stand for any term. This is the same problem that was discussed at the beginning of this chapter, in page 59: in both cases we have a term that is not a quotation of any other term.

Induction is very hard to get: there is no well-founded inductive structure on Term. All we have in the representation is abstract function so we need to find a way to restrict these functions to ones we know how to recurse on. This problem is indicated by a negative occurrence of Term on the left side of the `lam` constructor, which corresponds to the fact that we get as input some arbitrary term, and if it is used for anything except arguments to syntactic constructors, we get exotic terms. A simple definition on HOAS cannot 'penetrate' functions completely, leaving us with no base case. It is possible, however, to define an induction principle that is based on the concrete counterpart of substitution functions, see Theorem 5.2.21 (`SubstFunc` induction) in the following chapter.

Note that the '`aterm`' definition of Figure 4.1 suffers from the same problem: '`make_aterm`' can be used with arbitrary functions, including ones that are not proper substitution functions. This specific case could have been solved if the code would hide the '`make_aterm`' function and expose only '`term_to_aterm`' for creating '`aterm`' objects. '`term_to_aterm`' uses '`bterm_to_substfunc`' to convert a given variable list and a body term to a proper substitution function. Obviously, this solution can only work given the inefficient sample '`aterm`' implementation, which ultimately uses '`term`' functionality and all that it requires, saving nothing.

4.7.2 Survey of Proposed Solutions

In this section we briefly cover some of the proposed solutions within the HOAS community.

Using a Modal Type

The solution which is described by Despeyroux, Pfenning, and Schürmann [28] goes back to earlier work [23] for its motivation, where it was used to explain a run-time code generation system. It can be summarized as decomposing the $\mathsf{Term} \to \mathsf{Term}$ (primitive-recursive) function space into a modal operator and a parametric function space. The modal operator is a function over the meta-language's types, so that $(\Box\mathsf{Term}) \to \mathsf{Term}$ is the parametric function space. The intuition is that $\Box T$ is the type of values representing values in T: a rough equivalent for $\Box T$ is our '`is_subst`' predicate, when the type of the represented value is known.

Following this intuition, it is not surprising that the $\Box$ operator can be used to restrict values so it is possible to construct terms freely with `lam`, without getting exotic terms. This restriction works like '`is_subst`' in that it forbids functions from using their input values, except that this is enforced through the type system rather that through a lexical analysis of the functions. To get primitive recursion, a syntactic-like mechanism is used: two functions (H_a, H_l) are given, and the recursion result is calculated by replacing every occurrence of `app` and `lam` by H_a and H_l respectively. The result is limited, for example, the authors indicate that structural equality between two terms cannot be expressed using this. They hypothesize that a pattern-matching calculus might be a more practical solution.

Adding a Var Type

Facing the same two fundamental problems, Despeyroux, Felty, and Hirschowitz [27] offer a different approach: adding a new 'Var' type. This makes the syntax representation have another constructor for these objects, so the Term type is now defined as:

$$\begin{aligned}\mathsf{Term} ::= \quad & \texttt{var} : \mathsf{Var} \to \mathsf{Term} \\ \mid \quad & \texttt{app} : \mathsf{Term} \times \mathsf{Term} \to \mathsf{Term} \\ \mid \quad & \texttt{lam} : (\mathsf{Var} \to \mathsf{Term}) \to \mathsf{Term}\end{aligned}$$

For example, the Church "1" numeral representation becomes:

$$1 = \lambda f x.\ f x \quad \text{is represented by:} \quad \texttt{lam}(\lambda f.\ \texttt{lam}(\lambda x.\ \texttt{app}(\texttt{var}(f), \texttt{var}(x))))$$

Note that the actual type that implements Var can be anything, it can be as simple as natural numbers, symbols, or strings. The resulting representation of various syntax objects contain functions that expect this type, but it is not expected to serve any computational role.

By itself, this does not prevent exotic terms; we can get them using the same way, by using a function that inspects the value it gets as an input — restricted now to a variable rather than any term, but still results in a misbehaving function and an exotic term. For example, if Var is implemented as natural numbers:

$$\begin{aligned}\texttt{foo} \ &:= \ \lambda f, x.\ \texttt{case}\ x\ \texttt{of}\ 0 \Rightarrow \texttt{var}(x) \\ &\qquad\qquad \mid\ \texttt{succ}(n) \Rightarrow \texttt{app}(\texttt{var}(f), \texttt{var}(n)) \\ \texttt{exot}_1 \ &:= \ \texttt{lam}(\lambda f.\ \texttt{lam}(\lambda x.\ \texttt{foo}(f, x)))\end{aligned}$$

Two more types of exotic terms are discussed in this work [27]:

$$\begin{aligned}\texttt{exot}_2 \;&:=\; \texttt{lam}(\lambda x.\ \texttt{case}\ x\ \texttt{of}\ 0 \Rightarrow \texttt{var}(0)\\ &\qquad\qquad\qquad\qquad \mid\ \texttt{succ}(n) \Rightarrow \texttt{var}(\texttt{succ}(n))\\ \texttt{exot}_3 \;&:=\; \texttt{var}(\texttt{succ}(0))\end{aligned}$$

$\texttt{exot}_2$ demonstrates the need for using extensional equality, and $\texttt{exot}_3$ uses a free variable. The solution for these problems that are suggested in this work [27] is using a '`valid`' predicate which is similar to:

$$\texttt{valid} := \lambda e.\ \exists e'.\ \texttt{eq}(e, e')\ \&\ \texttt{valid}_0(e')$$

where '`eq`' is an extensional equality predicate, and '$\texttt{valid}_0$' is a predicate that can use some set of free variables. The '$\texttt{valid}_0$' predicate is constructed in a way that allows only non-exotic terms (which will disqualifies $\texttt{exot}_1$) using the extra Var type.

The main problem with this approach is that they it loses substitution: in the example of the beta reduction specification we previously had:

$$\texttt{app}(\texttt{lam}(f), t) \quad \text{reduces to:} \quad f(t)$$

But if, for example, we take f as $\lambda x.\ \texttt{app}(\texttt{var}(x), \texttt{var}(x))$, then for some v : Var we get $f(v) = \texttt{app}(\texttt{var}(v), \texttt{var}(v))$ where we actually want $\texttt{app}(v, v)$. This means that substitution needs to be defined as an operation that applies the function and replaces the resulting `var`s. This is further complicated by the fact that they cannot define such a substitution function directly, but through a predicate. Also, when they try to handle different variable types (*e.g.*, type variables), they encounter a need to add more Var types. In short, this approach keeps the automatic alpha-renaming property, but gives up on automatic substitution and similar functionality.

Additional Solutions

More solutions have been suggested in the HOAS community, for example, Hofmann [38] mentions Honsell and Miculan [39] who show that if the Var type of Despeyroux et al [27] is left unspecified (*e.g.*, as a variable of some Set type), then adequacy still holds. This goes well with the intuition for our '`is_subst`': the main point is forbidding any usage of input values more than passing them around to template holes. Using an unspecified Var type makes it impossible to use its values, achieving a similar restriction. After a survey of these solutions, Hofmann [38] continues to use functor categories to justify the adequacy of the modal logic solution. More recent works focus on general solutions for the recursion problem over higher-order data, improving the modal type, etc.

Yet another direction that was initiated by Gabbay and Pitts [58, 31], is based on Set theory and permutations. This approach was first used for the FreshML implementation [62]. Recently, Christian Urban et al [73, 72] has improved this

approach, mainly making it compatible with the axiom-of-choice (a problem in earlier nominal logic work by Pitts et al). Doing this, Urban has accomplished impressive results that are based on nominal logic; this work can be viewed as coinciding with our results — detailed in the following section.

4.7.3 Nominal Techniques

In Urban's work [73], an inductive set is constructed, such that it is bijective with the α-equated lambda-terms ($\mathsf{CTerm}//\alpha$, or Term in our notation). The main method that is used to create this set is using variable permutations, which form the basis of Urban's formalization. Like our work, Urban motivates his work by observing the technical difficulties that arise when dealing with concrete syntax.

Using Urban's formalism, reasoning about syntax with binders becomes quite easy, in Urban's words: "reasoning about this definition is very similar to informal reasoning on paper". Indeed, using his formalization, Urban has been able to implement in Isabelle/HOL the standard proofs for Church-Rosser and strong normalization. There are therefore some similarities in using Urban's formalization and ours, and both can be seen as a leap in expressive power for dealing with meta-mathematics. (Urban's work was brought to our attention in the final stages of writing this text.)

Despite the high-level similarity between our method and Urban's, the implementation uses very different techniques. In fact, the nominal approach which Urban uses as a foundation, is quite different than most other HOAS approaches that we have discussed so far. The basic building blocks that are used in the nominal approach — permutations — can be considered as a variation on variable renaming, except that instead of renaming concrete names they are permuted. Variable permutations are simpler than renaming since they preserve α-equivalence which require some additional mechanics in the former [71]. For example, consider the term 'λ`x.x(y)`', clearly, a naive renaming of '`x`' to '`y`' leads to an inappropriate capture of '`y`', yielding a term with a different meaning. On the other hand, if we use permutations, for example '`(x y)`' — a permutations that replaces '`x`' with '`y`' and '`y`' with '`x`' — then the result is 'λ`y.y(x)`'. The intuition is that whenever renaming leads to capture, permutations will re-use the 'renamed' identifier (which is usually thrown away) to rename identifiers that could be captured. This extra robustness of permutations, is what allows them to be used at a very low level: when a permutation is applied over a term, it is applied recursively on both its subterms *and* its binding positions, so in this view, a binding position is no different than a subterm. An additional indication of the 'low-levelness' of this approach is in the definition of *support*: the support of a term is defined as

$$\mathtt{supp}(x) \equiv \{a \mid \mathtt{inf}\{b \mid (a\ b) \bullet x \neq x\}\}$$

but Urban explains how to compute it in a simple way:

$$\mathtt{supp}(a) = \{a\},\quad \mathtt{supp}(t_1 t_2) = \mathtt{supp}(t_1) \cup \mathtt{supp}(t_2),\quad \mathtt{supp}(\lambda a.\ t) = \mathtt{supp}(t) \cup \{a\}$$

— note that the third rule specifies that the support of a term is the set of *all* atoms that occur in it — free or not.

Indeed, later on, *nominal abstraction* are introduced, which behave in a very similar way to terms modulo α-equality — but the definition of nominal abstractions use techniques that reminiscent of conventional renaming. For example, a side condition to one of the cases in the definition is: $a \neq b \wedge x_1 = (a\ b) \bullet x_2 \wedge a \# x_2$, and later on, when a subset of functions from $\mathbb{A}$ to Φ is created to define Λ_α, the expression for these functions contains: '$\texttt{if}\ b\#t\ \texttt{then}\ (a\ b) \bullet t\ \texttt{else er}$' — in both of these, the freshness constraint $(\cdot\#\cdot)$ essentially turns the permutation into a plain renaming with a new variable.

Given this intuition, it is not surprising that once nominal abstractions are defined, and the Λ_α is created using the subset of $\mathbb{A} \to \Phi$ functions, Urban is in a very similar setting to ours. Nominal abstractions behave like α-equivalence classes over concrete terms (when used with concrete syntax), and the created set of functions is essentially viewed as an encoding of such classes. For example, the support set of members of Λ_α behaves in a way that coincides with the usual meaning of free variables, compare these construction rules

$$\texttt{supp}(\texttt{am}(a)) = \{a\}, \quad \texttt{supp}(\texttt{pr}(t_1 t_2)) = \texttt{supp}(t_1) \cup \texttt{supp}(t_2),$$
$$\texttt{supp}(\texttt{se}([a].\ t)) = \texttt{supp}(t) - \{a\}$$

to the above and note that the last one accumulates only free variables. The similarity goes further: as is the case with our substitution functions, Urban notes that deciding whether two $\mathbb{A} \to \Phi$ functions are equal is impossible in a general, but it is possible over the defined subset, since the construction rules provide a simple way to compare two functions in a way that makes it similar to a comparison of two substitution functions given their concrete representation. Furthermore, Urban's quotation function, '$\texttt{q}$', is similar to other such functions in different HOAS formalizations, including ours (see Section 3.5.4), with the nominal abstraction construct being its distinguishing feature, and with the usual result [73, Lemma 9]: $t_1 \approx t_2$ iff $q(t_1) = q(t_2)$ (where '$\approx$' coincides with '$=_\alpha$' and the right-hand-side equality is over Λ_α).

Using Urban's formalization, induction is cheap — it 'falls off' as a by product of the definition of Λ_α. However, he explains that this is "not very convenient in practice", since induction is on the construction of Λ_α rather than on the term structure. A new induction principle is therefore devised, which not only allows structural induction, but is enhanced with an additional variable that stands for the "context" of the induction. This is one of the major points of Urban's work (further explored in [72]).

Finally, a schematic layout of the major components in Urban's formalization and the rough equivalents in ours is shown in Figure 4.2.

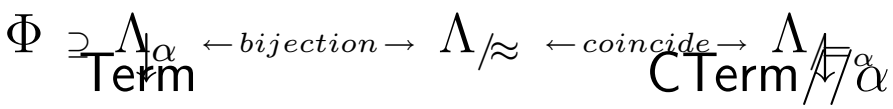


Figure 4.2: Major components in Urban's formalism; connections to ours

Comparison

The major issue with the nominal logic approach, is that it is not derived from simple syntax principles that are commonly used. This is evident in numerous terms that are specific to this approach and introduced during the presentation (Disagreement Set, Permutation Equality, PSets, Support, Freshness, Fs-PSets, Nominal Abstractions). This is in contrast to our work that is built on a foundation of plain concrete terms, and well-known alpha equality.

Compared to 'full HOAS' [57], Urban admits that mechanisms like capture-avoiding substitution do not come for free as in full HOAS (instead, his system "load[s] the work of such definitions onto the user"). On the other hand, an advantage is being able to deal with simultaneous substitutions with no difficulties. As we have seen, our formalization enjoys both of these advantages.

The nominal work has so far been focused on the lambda calculus, with a single binding form. This goes well with the uniform approach of permutations to syntax, regarding all sub-pieces uniformly, whereas Nuprl treats binding positions in a special way. Urban claims that there should not be any difficulties in expanding the method to deal with other languages, and mentions research that adapts the approach for the π-calculus, yet our approach deals with arbitrary operators thanks to Nuprl's uniform syntax representation and open-endedness.

A major advantage of the nominal logic approach is the fact that induction comes freely with the Λ_α construction. As we shall see in Chapter 5, we do require significant work to achieve an induction rule. The presented formalization is quite clean and elegant, and since the end result is quite similar to ours, we may have been able to use it to justify our system, had it been developed at the time our research was done.

The support for Barendregt-style proofs via a customized induction principle that carries around a 'variable context bag' seems like a useful feature for common syntax-oriented proof techniques. However, we suspect that the MetaPRL derivative of our work [55] can tackle this problem in a more elegant way, by using '`bterm`' as the basic syntax building blocks, considering a top-level '`bterm`' as denoting a '`w`'ith a given 'free variable context' (see [55] for further details).

Finally, Urban mentions a few problems that are due to the Isabelle/HOL environment, the main problem being that it requires injectivity of term constructors, but nominal abstractions are not. This leads to a work-around of "defining functions as inductive relations and then use the definite description operator THE of

Isabelle to turn relations into functions". A few examples demonstrate the complexity, for example, in proving "totality" of these relations that is required for turning them into functions. He also mentions that "ideally, a user just defines an inductive datatype and indicates where binders are—the rest of the infrastructure should be provided by the theorem prover", which might indicates a possible advantage of using Urban's work in the context of Nuprl, taking advantage of its syntax uniformity.

4.7.4 Comparison with the Nuprl Solution

The quotation system that we have devised in Nuprl is similar to standard HOAS. This holds for the representation as well as the problems we encounter. On the representation side, we have seen the three different 'lambda' operators in the right hand side of

$$\lambda fx.\ fx \quad \text{is represented by:} \quad \texttt{lam}(\lambda f.\ \texttt{lam}(\lambda x.\ \texttt{app}(f, x)))$$

and in Nuprl we also have

$$\lambda\texttt{f,x.apply(f;x)} \quad \text{is represented by:} \quad \underline{\lambda}\texttt{f,x}\underline{\texttt{.apply(f;x)}}$$

where the 'λ' in the HOAS case turns into the function semantics of a bound Nuprl subterm, and the other two correspond to the 'λ' operator id, and to '$\underline{\lambda}$', its shifted version. In addition, the quotation and unquotation functions that we have implemented in Nuprl ('`rquote_term`' and '`runquote_term`') convert a concrete term to another concrete term that represents it, *i.e.*, they are functions that work at the *concrete syntax* level (and use the relevant ML functionality), in an equivalent way to the HOAS translation that was discussed on page 73.

A technical difference between HOAS over some "conventional" syntax and our approach over Nuprl syntax, is that in the Nuprl case the syntax makes the binding structure explicit and uniform. This means that we don't have to encode syntax pieces that contain bindings using a term with a different binding structure, we just keep the same term structure. For example, the following translation (from [57]):

$$\ulcorner\texttt{let}\ v_1 = e_1\ \texttt{and}\ ...\ \texttt{and}\ v_n = e_n\ \texttt{in}\ e\urcorner \rightarrow \texttt{let}(\lambda v_1, ..., v_n.\ \ulcorner e\urcorner, \langle \ulcorner e_1\urcorner, ..., \ulcorner e_n\urcorner\rangle)$$

requires augmenting the Term type with a new `let` constructor, and massaging the original term into one that has all bindings in an explicit function. In the Nuprl case, there is no need for a specific constructor addition since we extend the system by assigning meaning to any operator with an '`rquote`' flag, so there is no need for changing the term binding structure (*e.g.*, the simple Nuprl '`let`' form is already in this form). This is indeed a technical point, but an important one since it was the motivating reason that made us reach our quotation mechanism, as we wanted quotations to use the same Nuprl terms so we can enjoy the benefits of string reflection.

A more substantial difference is the mechanism which we use to verify valid term representations (*i.e.*, ruling out exotic terms). Our approach is fundamentally different from standard HOAS solutions, in that it is more syntactic in its nature.

4.8 Conclusion

Our proof from Section 4.6.2 shows that the '`is_subst`' predicate is a valid restriction, based on pure mathematical principles, beginning with a clear definition of what can be considered a substitution function. These principles are founded on the well-known intuitive concept of concrete syntax. Our solution therefore uses only a few basic tools in addition to some syntax: functions, tuples, simple arithmetic, etc, as well as a syntax with an explicit uniform binding structure.

The resulting '`is_subst`' rule uses the syntactic structure of term-representing terms to verify that a syntactic construct is a valid Term— in itself, it does not depend on the type theory. Using this rule is easily accompanied with the existing environment: at any point during an '`is_subst`' proof it is possible to use existing type judgments, therefore making it possible to use not only literal quotations, but mixtures of quotations and descriptions. Such descriptions can make full use of the environment. Hence this rule can be used as an add-on with varying environments; in fact it is already in use in the MetaPRL framework [36] which uses an approach based on this work to implement syntactic reflection [55], and this rule is used to mix literal quotations and descriptions.

The same general principle can be used in programming languages, for example, Scheme's high-level macro facility, '`syntax-rules`' and MetaPRL's rewrite specifications are both specified as a template constructs that are inherently restricted to proper substitution functions. The various HOAS type-based approaches put this burden on a type system, but type checking code is a syntactic process as well. On the other hand, we enjoy the advantage of not being confined to a specific type system, or a specific type methodology.

A possible conjecture that explains the simplicity of our system compared with the heavy HOAS solutions, is that when a language revolves around a type system, it makes it more worthwhile to explain as much as possible using it, rather than escaping to new kinds of analyses. Nuprl's type system makes it easy to add new types based on any predicate (rule) using predicate subtypes, for our specific case, we can just take a subset type of all $\mathsf{Term}\ ^n\!\!\rightarrow \mathsf{Term}$ functions using the '`is_subst`' predicate:

$$\mathsf{SubstFunc}_n = \{f : \mathsf{Term}^n \!\rightarrow \mathsf{Term} \mid \mathtt{is\_subst}_n(f)\}$$

We believe that implementing our solution using a type system will lead to the same result (and the same kind of constraints), the question is whether the semantic explanation is simpler using our approach.

There is another observation that can be made when comparing our work to standard HOAS. Syntax-reflective implementation work involves languages at three different levels: a meta-language that implements an object language and that is

itself described in some theoretical language. This goes back to the philosophical discussion in Section 2.1, where the feature we want to reflect is bindings. We believe that our solution is elegant in that the three language levels are similar in nature, mainly because we require almost nothing more than the existence of binders and functions — in our case, the relation between the implementation's meta-language and the object-level language is validated using the (philosophically) similar concept of functions; in a sense, we have demonstrated HOAS based only on the mathematical concepts of a function, a language to describe functions, and substitution. It can therefore be considered as yet another step in the direction of bridging the gap between the logic world (where the meta-meta language is the central concept) and the programming language world (where the implementation language is central).

Chapter 5
Formalizing Representation

5.1 Design Constraints for a Representation Relation

Given the material that was presented so far, we want to verify that our quotation scheme can actually be used for a valid representation relation in Nuprl. For this we need to show that the following properties are satisfied:

(a) if $A \,\texttt{reps}\, B$ then A is closed (has no free variables).

(b) if $A \,\texttt{reps}\, B$ and $A \,\texttt{reps}\, B'$ then B is B'.

(c) if $A \,\texttt{reps}\, B$ and $A \cong C$ then $C \,\texttt{reps}\, B$ (here we assume that '$\cong$' is Howe's congruent extension of Kleene equality [41].)

(d) for every term B there is a term A such that $A \,\texttt{reps}\, B$.

(e) let $\texttt{rep}(B) \,\texttt{reps}\, B$ for all B, *i.e.*, let $\texttt{rep}(\cdot)$ be one of the functions implicit in d by the axiom of choice.
This assumption would be most naturally effected in our intended domain by defining the full quotation function and using it to find the standard representative.

(f) We add one more requirement, which is necessary if one is to be able to perform effective inquiries, such as testing term identity.
if $A \,\texttt{reps}\, B$ then A evaluates to some term.

5.2 Definitions, Facts, and Proofs

5.2.1 Description 1: Term type

Term is defined as the type of Nuprl terms, with the usual structure, modulo α-equivalence. The point where this is different than standard Nuprl formalism is that we always take this as the HOAS representation, as demonstrated in Chapter 4. The main difference between this and the Term definition from Chapter 4 is that we are now talking about actual Nuprl terms.

Note that this is not a definition — it is a description, as Term will be defined by material in this chapter (see Definition 5.2.6 (reps) below).

5.2.2 Definition 2: Atom type

Atom is defined as a subclass of Term — all terms with no subterms, *i.e.*,

$$\forall o : \mathsf{OpId}.\ \texttt{mkTerm}(o, 0, (\lambda i.\ ?), (\lambda i.\ ?)) \in \mathsf{Atom}$$

5.2.3 Definition 3: `RepsFormation` relation constructor

We define '`RepsFormation`' as a constructor for `reps`-like relations over terms:

$$
\begin{array}{l}
A\,\texttt{RepsFormation}_P\,B \equiv \\
\quad \exists\ o : \mathsf{OpId},\ k : \mathbb{N},\ a : (1...k \to \mathbb{N}),\ f, g : (i : 1...k \to \mathsf{SubstFunc}_{a_i}). \\
\qquad B = \texttt{mkTerm}(o, k, a, f) \\
\qquad \&\ A \underrightarrow{\texttt{evalsto}}_{\underline{o}} \texttt{mkTerm}(\underline{o}, k, a, g) \\
\qquad \&\ \forall i : 1...k,\ t : \mathsf{Atom}^{a_i}.\ g_i(\texttt{map}(\mathsf{q}, t))\ P\ f_i(t)
\end{array}
$$

where '$\underline{o}$' denotes the opid that is the shifted version of o, and 'q' is the quotation operation (see Definition 5.2.23 (q)). P will be used for a fixpoint in Theorem 5.2.5 (`RepsFormation` fixpoint).

5.2.4 Theorem 4: `RepsFormation` monotonic

It is simple to see that '`RepsFormation`' is monotonic:

$$\forall P, Q : \mathsf{Term}\text{-relation}.\ P \subseteq Q \Rightarrow \texttt{RepsFormation}_P \subseteq \texttt{RepsFormation}_Q$$

5.2.5 Theorem 5: `RepsFormation` fixpoint

Using Theorem 5.2.4 (`RepsFormation` monotonic):

$$
\begin{array}{l}
\exists Q : \mathsf{Term}\text{-relation}. \\
\quad \texttt{RepsFormation}_Q \subseteq Q \\
\quad \&\ \forall P : \mathsf{Term}\text{-relation}.\ \texttt{RepsFormation}_P \subseteq P \Rightarrow Q \subseteq P
\end{array}
$$

5.2.6 Definition 6: `reps` relation

'`reps`' is defined using Definition 5.2.3 (`RepsFormation`), as the Q in Theorem 5.2.5 (`RepsFormation` fixpoint):

$$\texttt{reps} \equiv \text{the strongest } P \text{ such that } \texttt{RepsFormation}_P \subseteq P$$

or:

$$
\begin{array}{l}
A\,\texttt{reps}\,B \equiv \\
\quad \exists o : \mathsf{OpId},\ k : \mathbb{N},\ a : (1...k \to \mathbb{N}),\ f, g : (i : 1...k \to \mathsf{SubstFunc}_{a_i}). \\
\qquad B = \texttt{mkTerm}(o, k, a, f) \\
\qquad \&\ A \underrightarrow{\texttt{evalsto}}_{\underline{o}} \texttt{mkTerm}(\underline{o}, k, a, g) \\
\qquad \&\ \forall i : 1...k,\ t : \mathsf{Atom}^{a_i}.\ g_i(\texttt{map}(\mathsf{q}, t))\,\texttt{reps}\,f_i(t)
\end{array}
$$

5.2.7 Theorem 7: `reps` induction

This follows from Theorem 5.2.5 (`RepsFormation` fixpoint) and Definition 5.2.6 (`reps`):

$$\forall P : \mathsf{Term}\text{-relation}.\ (\texttt{RepsFormation}_P \subseteq P) \Rightarrow \texttt{reps} \subseteq P$$

Another way to write this is:

$$\begin{array}{l}\forall P : \mathsf{Term}\text{-relation}.\\ \quad (\forall x, y : \mathsf{Term}.\ x\ \texttt{RepsFormation}_P\ y \Rightarrow x\ P\ y)\\ \qquad \Rightarrow (\forall x, y : \mathsf{Term}.\ x\ \texttt{reps}\ y \Rightarrow x\ P\ y)\end{array}$$

5.2.8 Theorem 8: `reps` is `RepsFormation` fixpoint

We need to show that

$$\texttt{RepsFormation}_{\texttt{reps}} = \texttt{reps}.$$

First, by Definition 5.2.6 (`reps`) we get:

$$\texttt{RepsFormation}_{\texttt{reps}} \subseteq \texttt{reps}$$

Using this with Theorem 5.2.4 (`RepsFormation` monotonic), we get:

$$\Rightarrow \texttt{RepsFormation}_{\texttt{RepsFormation}_{\texttt{reps}}} \subseteq \texttt{RepsFormation}_{\texttt{reps}}$$

and by Theorem 5.2.7 (`reps` induction):

$$\Rightarrow \texttt{reps} \subseteq \texttt{RepsFormation}_{\texttt{reps}}$$

We now need to show that the design constraints specified in Section 5.1 hold for this '`reps`'.

5.2.9 Theorem 9: `reps` closed (requirement (a))

$$A\ \texttt{reps}\ B \Rightarrow \texttt{closed}(A)$$

This is trivially true by the definition of $\underrightarrow{\texttt{evalsto}}$ which holds only for closed terms.

5.2.10 Theorem 10: `reps` evaluates (requirement (f))

$$A\ \texttt{reps}\ B \Rightarrow \exists t : \mathsf{Term}.\ A\ \underrightarrow{\texttt{evalsto}}\ t$$

This is true by Definition 5.2.6 (`reps`).

5.2.11 Theorem 11: `reps` unique (requirement (b))

$$A \,\texttt{reps}\, B \;\&\; A \,\texttt{reps}\, C \Rightarrow B = C$$

To prove this, begin with a complete version:

$$\forall A, B, C : \mathsf{Term}.\; A \,\texttt{reps}\, B \;\&\; A \,\texttt{reps}\, C \Rightarrow B = C$$

Change it to a form that fits Theorem 5.2.7 (`reps` induction):

$$\forall A, B : \mathsf{Term}.\; A \,\texttt{reps}\, B \Rightarrow \forall C : \mathsf{Term}.\; A \,\texttt{reps}\, C \Rightarrow B = C$$

Define R as the right hand side relation:

$$A \;R\; B \equiv \forall C : \mathsf{Term}.\; A \,\texttt{reps}\, C \Rightarrow B = C$$

and we want to show:

$$\forall A, B : \mathsf{Term}.\; A \,\texttt{reps}\, B \Rightarrow A \;R\; B$$

Using Theorem 5.2.7 (`reps` induction), it is enough to show:

$$\forall A, B : \mathsf{Term}.\; A \,\texttt{RepsFormation}_R\, B \Rightarrow A \;R\; B$$

Let A and B be two arbitrary Terms such that $A \,\texttt{RepsFormation}_R\, B$; we need to show $A \;R\; B$. Also let C be a Term such that $A \,\texttt{reps}\, C$, and we now need $B = C$. Expanding what we know on A, B, C:

$$\begin{aligned}
\exists o : \mathsf{OpId}, k : \mathbb{N}, a : (1...k \to \mathbb{N}), f, g : ((i : 1...k) \to \mathsf{SubstFunc}_{a_i}).\\
&B = \texttt{mkTerm}(o, k, a, f)\\
&\;\&\; A \,\underrightarrow{\texttt{evalsto}}\, \texttt{mkTerm}(\underline{o}, k, a, g)\\
&\;\&\; \forall i : 1...k,\; t : \mathsf{Atom}^{a_i}.\; g_i(\texttt{map}(\mathsf{q}, t)) \;R\; f_i(t)
\end{aligned}$$

and

$$\begin{aligned}
\exists o_1 : \mathsf{OpId}, k_1 : \mathbb{N}, a_1 : (1...k_1 \to \mathbb{N}), f_1, g_1 : ((i : 1...k_1) \to \mathsf{SubstFunc}_{a_{1i}}).\\
&C = \texttt{mkTerm}(o_1, k_1, a_1, f_1)\\
&\;\&\; A \,\underrightarrow{\texttt{evalsto}}\, \texttt{mkTerm}(\underline{o_1}, k_1, a_1, g_1)\\
&\;\&\; \forall i : 1...k_1,\; t : \mathsf{Atom}^{a_{1i}}.\; g_{1i}(\texttt{map}(\mathsf{q}, t)) \,\texttt{reps}\, f_{1i}(t)
\end{aligned}$$

By the uniqueness of the right hand side of $\underrightarrow{\texttt{evalsto}}$ we get:

$$o = o_1 \;\;\&\;\; k = k_1 \;\;\&\;\; a = a_1 \;\;\&\;\; g = g_1$$

So the above can be rewritten as:

$$\begin{aligned}
&\exists o : \mathsf{OpId}, k : \mathbb{N}, a : (1...k \to \mathbb{N}), f, g, f_1 : ((i : 1...k) \to \mathsf{SubstFunc}_{a_i}). \\
&\qquad B = \mathtt{mkTerm}(o, k, a, f) \\
&\qquad \& C = \mathtt{mkTerm}(o, k, a, f_1) \\
&\qquad \& A \xrightarrow[\underline{o}]{\mathtt{evalsto}} \mathtt{mkTerm}(\underline{o}, k, a, g) \\
&\qquad \& \forall i : 1...k,\ t : \mathsf{Atom}^{a_i}.\ g_i(\mathtt{map}(\mathtt{q}, t))\ R\ f_i(t) \qquad \text{(a)} \\
&\qquad\qquad \& \ g_i(\mathtt{map}(\mathtt{q}, t))\ \mathtt{reps}\ f_{1_i}(t) \qquad \text{(b)}
\end{aligned}$$

Now by the value of B and C, to show that $B = C$ we only need $f = f_1$ (under the last $\forall$ qualifier):

$$f_i(t) = f_{1_i}(t)$$

but expanding (a) we get:

$$\forall D : \mathsf{Term}.\ g_i(\mathtt{map}(\mathtt{q}, t))\ \mathtt{reps}\ D \Rightarrow f_i(t) = D$$

and from this, with (b):

$$g_i(\mathtt{map}(\mathtt{q}, t))\ \mathtt{reps}\ f_{1_i}(t) \Rightarrow f_i(t) = f_{1_i}(t)$$

QED.

5.2.12 Theorem 12: `reps` squiggle (requirement (c))

$$A\ \mathtt{reps}\ B\ \&\ A \cong C \Rightarrow C\ \mathtt{reps}\ B$$

This proof requires referencing Howe's congruent extension of Kleene equality, '$\cong$' [41]. The basic property that we use is that $\cong$-equality has a "same-shape" property — two $\cong$-equal terms evaluate to a terms with the same outermost (canonical) opid and arity, and substitution function bodies that produce $\cong$-equal results on $\cong$-equal inputs. Actually, we need a weaker version, where the substitutions produce $\cong$-equal results on equal canonical input atoms:

$$\begin{aligned}
&\forall A, B : \mathsf{Term}.\ A \cong B \Rightarrow \qquad (5.1) \\
&\quad \exists o : \mathsf{OpId},\ k : \mathbb{N},\ a : (1...k \to \mathbb{N}),\ f, g : ((i : 1...k) \to \mathsf{SubstFunc}_{a_i}). \\
&\qquad A \xrightarrow[\underline{o}]{\mathtt{evalsto}} \mathtt{mkTerm}(o, k, a, f) \\
&\qquad \& \ B \xrightarrow[\underline{o}]{\mathtt{evalsto}} \mathtt{mkTerm}(o, k, a, g) \\
&\qquad \& \ \forall i : 1...k,\ t : \mathsf{Atom}^{a_i}. \\
&\qquad\qquad (\forall j : 1...a_i.\ \mathtt{is_canonical}(t_i)) \Rightarrow f_i(t) \cong g_i(t)
\end{aligned}$$

The proof uses Theorem 5.2.7 (`reps` induction), similarly to Theorem 5.2.11 (`reps` unique). We begin with the complete version of what we want to prove:

$$\forall A, B, C : \mathsf{Term}.\ A\ \mathtt{reps}\ B\ \&\ A \cong C \Rightarrow C\ \mathtt{reps}\ B$$

Rephrasing it for Theorem 5.2.7 (`reps` induction):

$$\forall A, B : \mathsf{Term}.\ A\,\mathtt{reps}\,B \Rightarrow \forall C : \mathsf{Term}.A \cong C \Rightarrow C\,\mathtt{reps}\,B$$

Define R as the right hand side relation:

$$A\ R\ B \equiv \forall C : \mathsf{Term}.\ A \cong C \Rightarrow C\,\mathtt{reps}\,B$$

and we want to show:

$$\forall A, B : \mathsf{Term}.\ A\,\mathtt{reps}\,B \Rightarrow A\ R\ B$$

Using Theorem 5.2.7 (`reps` induction), it is enough to show:

$$\forall A, B : \mathsf{Term}.\ A\,\mathtt{RepsFormation}_R\,B \Rightarrow A\ R\ B$$

Let $A, B : \mathsf{Term}.\ A\,\mathtt{RepsFormation}_R\,B$, and we need to show

$$A\ R\ B.$$

Also let $C : \mathsf{Term}.\ A \cong C$, and we now need

$$C\,\mathtt{reps}\,B$$

Expanding what we know on A, B:

$$\begin{aligned}\exists o : \mathsf{OpId},\ k : \mathbb{N},\ a : (1...k \to \mathbb{N}),\ f, g : ((i : 1...k) \to \mathsf{SubstFunc}_{a_i}).\\ B = \mathtt{mkTerm}(o, k, a, f)\\ \&\ A\,\underline{\mathtt{evalsto}}_{\rightarrow}\,\mathtt{mkTerm}(\underline{o}, k, a, g)\\ \&\ \forall i : (1...k),\ t : \mathsf{Atom}^{a_i}.\ g_i(\mathtt{map}(\mathtt{q}, t))\ R\ f_i(t)\end{aligned}$$

and because C is $\cong$-equal to A, then by what A evaluates to, and by (5.1), we know more — since '$\mathtt{q}$' produces canonical terms then (adding new facts to the above):

$$\begin{aligned}\exists o : \mathsf{OpId},\ k : \mathbb{N},\ a : (1...k \to \mathbb{N}),\ f, g, g_1 : ((i : 1...k) \to \mathsf{SubstFunc}_{a_i}).\\ B = \mathtt{mkTerm}(o, k, a, f)\\ \&\ A\,\underline{\mathtt{evalsto}}_{\rightarrow}\,\mathtt{mkTerm}(\underline{o}, k, a, g)\\ \&\ C\,\underline{\mathtt{evalsto}}_{\rightarrow}\,\mathtt{mkTerm}(\underline{o}, k, a, g_1)\\ \&\ \forall i : (1...k),\ t : \mathsf{Atom}^{a_i}.\\ g_i(\mathtt{map}(\mathtt{q}, t))\ R\ f_i(t) \qquad (5.2)\\ g_i(\mathtt{map}(\mathtt{q}, t)) \cong g_{1_i}(\mathtt{map}(\mathtt{q}, t)) \qquad (5.3)\end{aligned}$$

And we need to show that

$$C\,\mathtt{reps}\,B$$

for which we have all the pieces except for one which needs showing (under the last $\forall$ qualifier):

$$g_{1_i}(\mathtt{map}(\mathtt{q}, t))\,\mathtt{reps}\,f_i(t)$$

but expanding (5.2) we get:

$$\forall D : \mathsf{Term}.\ g_i(\mathtt{map}(\mathtt{q}, t)) \cong D \Rightarrow D\,\mathtt{reps}\,f_i(t)$$

and from this, with (5.3):

$$g_i(\mathtt{map}(\mathtt{q}, t)) \cong g_{1_i}(\mathtt{map}(\mathtt{q}, t)) \Rightarrow g_{1_i}(\mathtt{map}(\mathtt{q}, t))\,\mathtt{reps}\,f_i(t)$$

QED.

In the above, the quotation function '`q`' was used but a definition was not given. Providing such a definition is also needed for the last basic '`reps`' property that we need — the existence of a '`rep`' function that, given a term, returns another that represents it (we will later show that '`q`' can be used as such a function). Defining '`q`' requires induction on HOAS terms, so we should describe this first. To do this we need to inspect how HOAS terms are constructed from and destructed to smaller terms.

With concrete terms, the situation is simple. A term is constructed from an operator id, a list of parameters (which we ignore for simplicity), and a list of bound subterms. A bound subterm is built from a list of variables and a term which (possibly) contains free occurrences of these variables. In the HOAS representation, substitution functions are the equivalent of bound subterms, as they carry a body and bindings in a functional form. The problem is that we cannot construct such substitution functions using free variables without considerable work (that most likely involves translating from concrete to abstract terms and back). Because of this, we construct substitution functions from other such substitution functions — the concrete analogy is that we define bound terms using bound terms and skip the intermediate term. This is achieved by a '`mkSubstFunc`' constructor that gets an arity (representing the arity of the resulting function), an operator id, and a list of substitution functions of at least that arity. For example, using "$\langle\cdot\rangle$" to denote concrete bound term syntax for substitution functions:

$$\mathtt{mkSubstFunc}_1(\text{``}\mathtt{foo}\text{''}, [\langle x.\ x\rangle; \langle y, x.\ y + x\rangle]) = \langle x.\ \mathtt{foo}(x;\ x_1.\ x + x_1)\rangle$$

This seems redundant as it is very similar to creating a term — we could define '`mkSubstFunc`' over a term, for example, changing the above to:

$$\mathtt{mkSubstFunc}_1(\langle \mathtt{foo}(x.x; y, x.y + x)\rangle) = \langle x.\ \mathtt{foo}(x;\ x_1.\ x + x_1)\rangle$$

but this is more complicated since it requires discriminating terms based on the arities of their substitution functions.

5.2.13 Definition 13: `mkSubstFunc` constructor

The definition of '`mkSubstFunc`' is therefore:

$$\forall n : \mathbb{N},\ o : \mathsf{OpId},\ k : \mathbb{N},\ a : (1...k \to \mathbb{N}),\ f : ((i : 1...k) \to \mathsf{SubstFunc}_{n+a_i}).$$
$$\texttt{mkSubstFunc}_n(o, k, a, f) \equiv$$
$$\lambda t : \mathsf{Term}^n.\ \texttt{mkTerm}(o, k, a, \lambda i : 1...k.\ \lambda s : \mathsf{Term}^{a_i}.\ f_i(\mathsf{append}(t, s)))$$

Note that this is a way to create a substitution function from other substitution functions. The base case of "simple" substitution functions is using simple projections.

5.2.14 Theorem 14: `mkSubstFunc` generates substitutions

The claim here is that given substitution functions, '`mkSubstFunc`' generates a substitution function:

$$\forall n : \mathbb{N},\ o : \mathsf{OpId},\ k : \mathbb{N},\ a : (1...k \to \mathbb{N}),\ f : ((i : 1...k) \to \mathsf{SubstFunc}_{n+a_i}).$$
$$\texttt{mkSubstFunc}_n(o, k, a, f) \in \mathsf{SubstFunc}_n$$

for this, it is enough to show:

$$\forall n : \mathbb{N},\ o : \mathsf{OpId},\ k : \mathbb{N},\ a : (1...k \to \mathbb{N}),\ f : ((i : 1...k) \to \mathsf{SubstFunc}_{n+a_i}).$$
$$\texttt{is\_subst}_n(\texttt{mkSubstFunc}_n(o, k, a, f))$$

Using the main theorem of Chapter 4 (Section 4.6.2), this is simple if we just shuffle things a bit. Given n, o, k, a, and f as specified, we define:

$$g : (\mathsf{Term}^n \to i : 1...k \to \mathsf{SubstFunc}_{a_i}) \equiv \lambda t, i, s.\ f_i(\mathsf{append}(t, s))$$

It is obvious that this definition has the correct type, and in addition:

$$\forall i : (1...k).\ \lambda_{(n)} t, s.\ g(t)(i)(s) = \lambda_{(n)} t, s.\ f_i(\mathsf{append}(t, s)) = f_i$$

so according to the '`is_subst`' theorem, we get:

$$\begin{aligned}
&\texttt{is_subst}_n(\texttt{mkTerm}(o, k, a) \circ g)\\
&\quad \Rightarrow \texttt{is_subst}_n(\lambda t : \mathsf{Term}^n.\ (\texttt{mkTerm}(o, k, a) \circ g)(t))\\
&\quad \Rightarrow \texttt{is_subst}_n(\lambda t : \mathsf{Term}^n.\ \texttt{mkTerm}(o, k, a)(g(t)))\\
&\quad \Rightarrow \texttt{is_subst}_n(\lambda t : \mathsf{Term}^n.\ \texttt{mkTerm}(o, k, a)(\lambda i, s.\ f_i(\mathsf{append}(t, s))))\\
&\quad \Rightarrow \texttt{is_subst}_n(\lambda t : \mathsf{Term}^n.\ \texttt{mkTerm}(o, k, a, \lambda i, s.\ f_i(\mathsf{append}(t, s))))\\
&\quad \Rightarrow \texttt{is_subst}_n(\texttt{mkSubstFunc}_n(o, k, a, f))
\end{aligned}$$

QED.

5.2.15 Definition 15: `mkTerm` constructor

For the sake of completeness, this is the type of '`mkTerm`' from Chapter 4:

$$\forall o : \mathsf{OpId},\ k : \mathbb{N},\ a : (1...k \to \mathbb{N}),\ f : ((i : 1...k) \to \mathsf{SubstFunc}_{a_i}).$$
$$\texttt{mkTerm}(o, k, a, f) in \mathsf{Term}$$

In Chapter 4, the actual definition of '`mkTerm`' was done using a translation of the concrete '`mkCTerm`'. Here this is not necessary: terms are created directly using substitution functions. The relation between '`mkTerm`' and '`mkSubstFunc`' is simple using a function of a zero-size tuple:

$$\forall o : \mathsf{OpId},\ k : \mathbb{N},\ a : (1...k \to \mathbb{N}),\ f : ((i : 1...k) \to \mathsf{SubstFunc}_{a_i}).$$
$$\texttt{mkSubstFunc}_0(o, k, a, f) = \lambda\ .\ \texttt{mkTerm}(o, k, a, f)$$

5.2.16 Definition 16: `sizeof` operator

We define an auxiliary '`sizeof`' function that measures sizes of various objects. This relies on the ability to convert any substitution function to a term and a list of variables (as discussed in Chapter 4) — name this '`sf_to_subst`':

$$\forall n : \mathbb{N},\ f : \mathsf{SubstFunc}_n.\ \exists v : \mathsf{CVar}^n, b : \mathsf{CTerm}.$$
$$\texttt{sf\_to\_subst}_n(f) = \langle v, b\rangle\ \ \&\ \ f = \lambda t : \mathsf{Term}^n.\ b[t\lfloor/v]\rceil$$

And now '`sizeof`' is defined as an overloaded function, with the '`SubstFunc`' version using the concrete version:

$$\forall n : \mathbb{N},\ f : \mathsf{SubstFunc}_n.$$
$$\text{let } \langle v, b\rangle = \texttt{sf\_to\_subst}_n(f) \text{ in } \texttt{sizeof}_n(f) = \texttt{sizeof}(v, b)$$
$$\forall n : \mathbb{N}, v : \mathsf{CVar}^n, b : \mathsf{CTerm}.$$
$$\texttt{sizeof}(v, b) = \text{if } (\exists i : (1...n).\ b = v_i) \text{ then } 0$$
$$\text{else } 1 + \Sigma\,\texttt{map}(\lambda v_1, b_1.\ \texttt{sizeof}(\texttt{append}(v, v_1), b_1),$$
$$\texttt{bterms\_of\_term}(b))$$

Intuitively, the '`sizeof`' function counts the number of operator ids, not including bound variable terms (all opids that need to be shifted when quoting).

5.2.17 Corollary 17: alpha-renaming preserves size

According to Definition 5.2.16 (`sizeof`), it is easy to see that modifying bound names yields a result with the same size:

$$\forall n : \mathbb{N},\ v_1, v_2 : \mathsf{CVar}^n,\ b_1, b_2 : \mathsf{BndCTerm}.$$
$$b_1 =_\alpha b_2[v_1/v_2]\ \Rightarrow\ \texttt{sizeof}(v_1, b_1) = \texttt{sizeof}(v_2, b_2)$$

This is what allows '`sizeof`' to be defined on abstract terms in the first place.

5.2.18 Theorem 18: SubstFunc is mkSubstFunc or a projection

We now claim that any substitution function (SubstFunc, achieved by some concrete term and a list of variables) can be constructed as either a projection function or by some '`mkSubstFunc`' application using "smaller" sub-substitution functions.

$$\begin{array}{l}\forall n : \mathbb{N},\ v : \mathsf{CVar}^n,\ b : \mathsf{CTerm}.\\ \quad \text{let } f = \lambda t : \mathsf{Term}^n.\ b[t\lfloor/v]\rceil \text{ in}\\ \quad\quad \exists i : 1...n.\ f = \pi^i_n\\ \quad\quad \text{or } \exists o : \mathsf{OpId},\ k : \mathbb{N},\ a : (1...k \to \mathbb{N}),\ g : ((i : 1...k) \to \mathsf{SubstFunc}_{n+a_i}).\\ \quad\quad\quad f = \mathtt{mkSubstFunc}_n(o, k, a, g)\end{array}$$

Assume n, v, and b given as specified. If $\exists i : 1...n.\ b = v_i$ then it is obvious that $f = \pi^i_n$ and we're done. So assuming that b is not among the vs, it is enough to show:

$$\begin{array}{c}\exists o : \mathsf{OpId},\ k : \mathbb{N},\ a : (1...k \to \mathbb{N}),\ g : ((i : 1...k) \to \mathsf{SubstFunc}_{n+a_i}).\\ \lambda t : \mathsf{Term}^n.\ b[t\lfloor/v]\rceil = \mathtt{mkSubstFunc}_n(o, k, a, g)\end{array}$$

Since b is a concrete term, we know:

$$\begin{array}{l}\exists o : \mathsf{OpId},\ k : \mathbb{N},\ a : (1...k \to \mathbb{N}),\\ \quad v_1 : ((i : 1...k) \to \mathsf{CVar}^{a_i}),\ b_1 : ((i : 1...k) \to \mathsf{CTerm}).\\ \quad\quad b = \mathtt{mkCTerm}(o, k, a, \lambda i : 1...k.\ \mathtt{mkBndCTerm}(v_{1i}, b_{1i}))\end{array}$$

In case some of the v_{1i} occur in v, we can simply rename them to other variables and the above turns to an α-equality:

$$\begin{array}{l}\exists o : \mathsf{OpId},\ k : \mathbb{N},\ a : (1...k \to \mathbb{N}),\\ \quad v_1 : ((i : 1...k) \to \mathsf{CVar}^{a_i}),\ b_1 : ((i : 1...k) \to \mathsf{CTerm}).\\ \quad\quad b =_\alpha \mathtt{mkCTerm}(o, k, a, \lambda i : 1...k.\ \mathtt{mkBndCTerm}(v_{1i}, b_{1i})) \qquad (5.4)\\ \quad\quad \&\ \forall i : 1...k.\ \forall j : 1...a_i,\ k : 1...n.\ v_{1i,j} \neq v_k \qquad (5.5)\end{array}$$

Using these,

$$\forall i : 1...k.(\lambda t : \mathsf{Term}^{n+a_i}.\ b_{1i}[t\lfloor/\mathtt{append}(v, v_{1i})]\rceil) \in \mathsf{SubstFunc}_{n+a_i}$$

so let:

$$g = \lambda i : 1...k.\ \lambda t : \mathsf{Term}^{n+a_i}.\ b_{1i}[t\lfloor/\mathtt{append}(v, v_{1i})]\rceil$$

It is now enough to show:

$$\lambda t : \mathsf{Term}^n.\ b[t\lfloor/v]\rceil = \mathtt{mkSubstFunc}_n(o, k, a, g)$$

According to Definition 5.2.13 (`mkSubstFunc`), this goal expands to:

$$\begin{aligned}&\lambda t : \mathsf{Term}^n.\ b[t\lfloor/v]\rceil \\ &\quad = \lambda t : \mathsf{Term}^n.\ \mathtt{mkTerm}(o, k, a, \lambda i : 1...k.\ \lambda s : \mathsf{Term}^{a_i}.\ g_i(\mathtt{append}(t, s)))\end{aligned}$$

so it is enough to show:

$$\begin{aligned}&\forall t : \mathsf{Term}^n. \\ &\quad b[t\lfloor/v]\rceil = \mathtt{mkTerm}(o, k, a, \lambda i : 1...k.\ \lambda s : \mathsf{Term}^{a_i}.\ g_i(\mathtt{append}(t, s)))\end{aligned}$$

Letting t as specified, and using (5.4) above, it is enough to show:

$$\begin{aligned}&\mathtt{mkCTerm}(o, k, a, \lambda i : 1...k.\ \mathtt{mkBndCTerm}(v_{1i}, b_{1i}))[t\lfloor/v]\rceil \\ &\quad = \mathtt{mkTerm}(o, k, a, \lambda i : 1...k.\ \lambda s : \mathsf{Term}^{a_i}.\ g_i(\mathtt{append}(t, s)))\end{aligned}$$

Note that it is okay to use (5.4) here — it is an alpha equality, but it is used inside an '$\cdot\rceil$' where substituting alpha-equivalent terms does not change the result.

According to the definition of '`mkTerm`' from Chapter 4, this turns to:

$$\begin{aligned}&\mathtt{mkCTerm}(o, k, a, \lambda i : 1...k.\ \mathtt{mkBndCTerm}(v_{1i}, b_{1i}))[t\lfloor/v]\rceil \\ &\quad = \mathtt{mkCTerm}(o, k, a, \lambda i : 1...k.\ \text{let } x = \text{newvar}_{a_i}(\lambda s : \mathsf{Term}^{a_i}.\ g_i(\mathtt{append}(t, s))) \text{ in} \\ &\qquad\qquad \mathtt{mkBndCTerm}(x\lfloor, g_i(\mathtt{append}(t, x))\lfloor))\rceil\end{aligned}$$

which is:

$$\begin{aligned}&\mathtt{mkCTerm}(o, k, a, \lambda i : 1...k.\ \mathtt{mkBndCTerm}(v_{1i}, b_{1i}))[t\lfloor/v] \\ &\quad =_\alpha \mathtt{mkCTerm}(o, k, a, \lambda i : 1...k.\ \text{let } x = \text{newvar}_{a_i}(\lambda s : \mathsf{Term}^{a_i}.\ g_i(\mathtt{append}(t, s))) \text{ in} \\ &\qquad\qquad \mathtt{mkBndCTerm}(x\lfloor, g_i(\mathtt{append}(t, x))\lfloor))\end{aligned}$$

according to the definition of g, this becomes:

$$\begin{aligned}&\mathtt{mkCTerm}(o, k, a, \lambda i : 1...k.\ \mathtt{mkBndCTerm}(v_{1i}, b_{1i}))[t\lfloor/v] \\ &\quad =_\alpha \mathtt{mkCTerm}(o, k, a, \lambda i : 1...k.\ \text{let } x = \text{newvar}_{a_i}(\lambda s : \mathsf{Term}^{a_i}.\ g_i(\mathtt{append}(t, s))) \text{ in} \\ &\qquad\qquad \mathtt{mkBndCTerm}(x\lfloor, \\ &\qquad\qquad\qquad b_{1i}[\mathtt{append}(t, x)\lfloor/\mathtt{append}(v, v_{1i})]\!\!\lfloor))\end{aligned}$$

and using $\star 2$ (pp. 62), we get:

$$\begin{aligned}&\mathtt{mkCTerm}(o, k, a, \lambda i : 1...k.\ \mathtt{mkBndCTerm}(v_{1i}, b_{1i}))[t\lfloor/v] \\ &\quad =_\alpha \mathtt{mkCTerm}(o, k, a, \lambda i : 1...k.\ \text{let } x = \text{newvar}_{a_i}(\lambda s : \mathsf{Term}^{a_i}.\ g_i(\mathtt{append}(t, s))) \text{ in} \\ &\qquad\qquad \mathtt{mkBndCTerm}(x\lfloor, \\ &\qquad\qquad\qquad b_{1i}[\mathtt{append}(t, x)\lfloor/\mathtt{append}(v, v_{1i})]))\end{aligned}$$

Since we assume that b (the '`mkCTerm`' on the left hand side) is not one of the vs, the substitution simply goes down recursively. Note that this is true even if b is

some other variable, since it means that substituting in it will leave it unchanged. So, beginning with the left hand side (and rewriting using alpha-equalities):

$$\texttt{mkCTerm}(o, k, a, \lambda i : 1...k.\ \texttt{mkBndCTerm}(v_{1i}, b_{1i}))[t\lfloor/v]$$

we first need to do the appropriate renamings. We need new variables y that do not occur in b_1 except for existing occurrences of v_1. We also need them to be disjoint from v and t to be able to push the other substitution inside later. So the left hand side is rewritten into:

$$\begin{array}{l}\texttt{mkCTerm}(o, k, a,\\ \qquad \lambda i : 1...k.\ \text{let } y = \text{newvar}_{a_i}(b_{1i}[0/v_{1i}]\lceil, t, v\lceil) \text{ in}\\ \qquad\qquad \texttt{mkBndCTerm}(y\lfloor, b_{1i}[y\lfloor/v_{1i}])\\ \qquad)[t\lfloor/v]\end{array}$$

But this will turn out to be overly restrictive — instead of choosing variables that do not occur in all ts, we could just choose ones that do not occur in ts that survive when it is substituting the vs into the b_{1i} body:

$$\begin{array}{l}\texttt{mkCTerm}(o, k, a,\\ \qquad \lambda i : 1...k.\ \text{let } y = \text{newvar}_{a_i}(b_{1i}[\texttt{append}(t, 0)\lfloor/\texttt{append}(v, v_{1i})]\lceil, v\lceil) \text{ in}\\ \qquad\qquad \texttt{mkBndCTerm}(y\lfloor, b_{1i}[y\lfloor/v_{1i}])\\ \qquad)[t\lfloor/v]\end{array}$$

We can see now that:

$$\begin{array}{l}\forall i, k.\ \text{let } y = \text{newvar}_{a_i}(b_{1i}[\texttt{append}(t, 0)\lfloor/\texttt{append}(v, v_{1i})]\lceil, v\lceil) \text{ in}\\ \qquad v_k\ \texttt{free\_in}\ b_{1i}[y\lfloor/v_{1i}] \Rightarrow \text{ none of } y\lfloor\ \texttt{free\_in}\ t_k\lfloor\ \texttt{or}\ v_k\end{array}$$

so we can use $\star 10$ (pp. 65) and rewrite this as:

$$\begin{array}{l}\texttt{mkCTerm}(o, k, a,\\ \qquad \lambda i : 1...k.\ \text{let } y = \text{newvar}_{a_i}(b_{1i}[\texttt{append}(t, 0)\lfloor/\texttt{append}(v, v_{1i})]\lceil, v\lceil) \text{ in}\\ \qquad\qquad \texttt{mkBndCTerm}(y\lfloor, b_{1i}[y\lfloor/v_{1i}][t\lfloor/v]))\end{array}$$

From (5.5) we know that v and each of the v_1 are disjoint. Together with the way 'y' was chosen, this means that we can do the substitutions in parallel:

$$\begin{array}{l}\texttt{mkCTerm}(o, k, a,\\ \qquad \lambda i : 1...k.\ \text{let } y = \text{newvar}_{a_i}(b_{1i}[\texttt{append}(t, 0)\lfloor/\texttt{append}(v, v_{1i})]\lceil, v\lceil) \text{ in}\\ \qquad\qquad \texttt{mkBndCTerm}(y\lfloor, b_{1i}[\texttt{append}(y\lfloor, t\lfloor)/\texttt{append}(v_{1i}, v)]))\end{array}$$

It is now possible to change the order and get the '$\cdot\lfloor$' outside the '`append`':

$$\begin{array}{l}\texttt{mkCTerm}(o, k, a,\\ \qquad \lambda i : 1...k.\ \text{let } y = \text{newvar}_{a_i}(b_{1i}[\texttt{append}(t, 0)\lfloor/\texttt{append}(v, v_{1i})]\lceil, v\lceil) \text{ in}\\ \qquad\qquad \texttt{mkBndCTerm}(y\lfloor, b_{1i}[\texttt{append}(t, y)\lfloor/\texttt{append}(v, v_{1i})]))\end{array}$$

Now we can see that the following holds:

$$\begin{array}{l}\text{let } z = \text{newvar}_{a_i}(b_{1i}[\texttt{append}(t,0)\lfloor/\texttt{append}(v,v_{1i})]\lceil) \text{ in}\\ \quad \text{newvar}_{a_i}(b_{1i}[\texttt{append}(t,0)\lfloor/\texttt{append}(v,v_{1i})]\lceil, v\lceil)\\ \quad\quad \texttt{not\_free\_in } b_{1i}[\texttt{append}(t,z)\lfloor/\texttt{append}(v,v_{1i})][0/z]\end{array}$$

(substituting 0 for z is simple: z does not occur in $b_{1i}[\texttt{append}(t,0)\lfloor/\texttt{append}(v,v_{1i})]$ so there is just one place to substitute.)

Because of this, we can do this renaming:

$$\begin{array}{l}\texttt{mkCTerm}(o,k,a,\\ \quad \lambda i:1...k.\ \text{let } y = \text{newvar}_{a_i}(b_{1i}[\texttt{append}(t,0)\lfloor/\texttt{append}(v,v_{1i})]\lceil, v\lceil) \text{ in}\\ \quad\quad \texttt{mkBndCTerm}(y\lfloor, b_{1i}[\texttt{append}(t,y)\lfloor/\texttt{append}(v,v_{1i})]))\\ =_\alpha\\ \texttt{mkCTerm}(o,k,a,\\ \quad \lambda i:1...k.\ \text{let } y = \text{newvar}_{a_i}(b_{1i}[\texttt{append}(t,0)\lfloor/\texttt{append}(v,v_{1i})]\lceil, v\lceil) \text{ and}\\ \quad\quad \text{let } z = \text{newvar}_{a_i}(b_{1i}[\texttt{append}(t,0)\lfloor/\texttt{append}(v,v_{1i})]\lceil) \text{ in}\\ \quad\quad\quad \texttt{mkBndCTerm}(z\lfloor, b_{1i}[\texttt{append}(t,y)\lfloor/\texttt{append}(v,v_{1i})][z/y]))\end{array}$$

but since y does not occur free in $b_{1i}[\texttt{append}(t,0)\lfloor/\texttt{append}(v,v_{1i})]$, substituting it is simple, again:

$$\begin{array}{l}\texttt{mkCTerm}(o,k,a,\\ \quad \lambda i:1...k.\ \text{let } y = \text{newvar}_{a_i}(b_{1i}[\texttt{append}(t,0)\lfloor/\texttt{append}(v,v_{1i})]\lceil, v\lceil) \text{ and}\\ \quad\quad \text{let } z = \text{newvar}_{a_i}(b_{1i}[\texttt{append}(t,0)\lfloor/\texttt{append}(v,v_{1i})]\lceil) \text{ in}\\ \quad\quad\quad \texttt{mkBndCTerm}(z\lfloor, b_{1i}[\texttt{append}(t,y)\lfloor/\texttt{append}(v,v_{1i})][z/y]))\\ =_\alpha\\ \texttt{mkCTerm}(o,k,a,\\ \quad \lambda i:1...k.\ \text{let } y = \text{newvar}_{a_i}(b_{1i}[\texttt{append}(t,0)\lfloor/\texttt{append}(v,v_{1i})]\lceil, v\lceil) \text{ and}\\ \quad\quad \text{let } z = \text{newvar}_{a_i}(b_{1i}[\texttt{append}(t,0)\lfloor/\texttt{append}(v,v_{1i})]\lceil) \text{ in}\\ \quad\quad\quad \texttt{mkBndCTerm}(z\lfloor, b_{1i}[\texttt{append}(t,z)\lfloor/\texttt{append}(v,v_{1i})]))\\ =_\alpha\end{array}$$

$$\begin{array}{l}\texttt{mkCTerm}(o,k,a,\\ \quad \lambda i:1...k.\ \text{let } z = \text{newvar}_{a_i}(b_{1i}[\texttt{append}(t,0)\lfloor/\texttt{append}(v,v_{1i})]\lceil) \text{ in}\\ \quad\quad \texttt{mkBndCTerm}(z\lfloor, b_{1i}[\texttt{append}(t,z)\lfloor/\texttt{append}(v,v_{1i})]))\end{array}$$

Now, getting back to the main goal, we need to show:

$$\begin{array}{l}\texttt{mkCTerm}(o, k, a,\\ \qquad \lambda i : 1...k.\ \text{let } z = \text{newvar}_{a_i}(b_{1i}[\texttt{append}(t, 0)\lfloor/\texttt{append}(v, v_{1i})]\rceil) \text{ in}\\ \qquad\qquad \texttt{mkBndCTerm}(z\lfloor, b_{1i}[\texttt{append}(t, z)\lfloor/\texttt{append}(v, v_{1i})]))\\ =_\alpha\\ \texttt{mkCTerm}(o, k, a,\\ \qquad \lambda i : 1...k.\ \text{let } x = \text{newvar}_{a_i}(\lambda s : \mathsf{Term}^{a_i}.\ g_i(\texttt{append}(t, s))) \text{ in}\\ \qquad\qquad \texttt{mkBndCTerm}(x\lfloor, b_{1i}[\texttt{append}(t, x)\lfloor/\texttt{append}(v, v_{1i})]))\end{array}$$

But by the definition of 'newvar$(\cdot)$' from Chapter 4, and the definition of g:

$$\begin{array}{l}\forall i : 1...k.\\ \quad \begin{array}{rl} x &= \text{newvar}_{a_i}(\lambda s : \mathsf{Term}^{a_i}.\ g_i(\texttt{append}(t, s)))\\ &= \text{newvar}_{a_i}(g_i(\texttt{append}(t, 0)))\\ &= \text{newvar}_{a_i}(b_{1i}[\texttt{append}(t, 0)\lfloor/\texttt{append}(v, v_{1i})]\rceil)\\ &= z\end{array}\end{array}$$

QED.

5.2.19 Corollary 19: Zero size is projection

The proof of both sides of the following is trivial:

$$\forall n : \mathbb{N}, f : \mathsf{SubstFunc}_n.\ \texttt{sizeof}_n(f) = 0 \ \Leftrightarrow\ \exists i : 1...n.\ f = \pi^i_n$$

5.2.20 Corollary 20: Positive size is mkSubstFunc

A fact which is the key for Theorem 5.2.21 (SubstFunc induction) below is:

$$\begin{array}{l}\forall n : \mathbb{N}, f : \mathsf{SubstFunc}_n.\\ \quad \texttt{sizeof}_n(f) > 0\\ \quad \Leftrightarrow \exists o : \mathsf{OpId}, k : \mathbb{N}, a : (1...k \to \mathbb{N}), g : ((i : 1...k) \to \mathsf{SubstFunc}_{n+a_i}).\\ \qquad\qquad f = \texttt{mkSubstFunc}_n(o, k, a, g)\\ \qquad\qquad \&\ \texttt{sizeof}_n(f) = 1 + \Sigma_{i=1}^k\, \texttt{sizeof}_{n+a_i}(g_i)\end{array}$$

The ($\Leftarrow$) direction is trivial. Assume n and f given with a positive size, according to Chapter 4 and Definition 5.2.16 (sizeof), and because f is not a projection, we know:

$$\begin{array}{l}\exists v : CVar^n, b : \mathsf{CTerm}.\\ \quad f = \lambda t : \mathsf{Term}^n.\ b[t\lfloor/v]\rceil\\ \quad \&\ sizeof_n(f) = sizeof(v, b)\\ \quad \&\ b\ \texttt{not\_in}\ v\end{array}$$

So according to Theorem 5.2.18 (`SubstFunc` is `mkSubstFunc` or a projection),

$$\exists o : \mathsf{OpId}, k : \mathbb{N}, a : (1...k \to \mathbb{N}), g : ((i : 1...k) \to \mathsf{SubstFunc}_{n+a_i}).$$
$$f = \texttt{mkSubstFunc}_n(o, k, a, g)$$

and it is easy to see that the size argument holds as well by the construction of Theorem 5.2.18 (`SubstFunc` is `mkSubstFunc` or a projection) and Definition 5.2.16 (`sizeof`).

5.2.21 Theorem 21: `SubstFunc` induction

Similarly to Definition 5.2.13 (`mkSubstFunc`), induction is parameterized by the arity of the function:

$$\begin{aligned}
&\forall P : (n : \mathbb{N} \to \mathsf{SubstFunc}_n \to Prop).\\
&\quad (\forall n : \mathbb{N}.\\
&\qquad \forall i : 1...n.\ P_n(\pi_n^i)\\
&\qquad \& \ \forall o : \mathsf{OpId}, k : \mathbb{N}, a : (1...k \to \mathbb{N}), f : ((i : 1...k) \to \mathsf{SubstFunc}_{n+a_i}).\\
&\qquad\quad (\forall i : 1...k.\ P_{n+a_i}(f_i)) \Rightarrow P_n(\texttt{mkSubstFunc}_n(o, k, a, f)))\\
&\quad \Rightarrow (\forall n : \mathbb{N}, f : \mathsf{SubstFunc}_n.\ P_n(f))
\end{aligned}$$

Proof: let $P : (n : \mathbb{N} \to \mathsf{SubstFunc}_n \to \mathbb{P})$ and assume:

$$\begin{aligned}
&\forall n : \mathbb{N}.\\
&\quad \forall i : 1...n.\ P_n(\pi_n^i) && (5.6)\\
&\quad \& \ \forall o : \mathsf{OpId}, k : \mathbb{N}, a : (1...k \to \mathbb{N}), f : ((i : 1...k) \to \mathsf{SubstFunc}_{n+a_i}). && (5.7)\\
&\qquad (\forall i : 1...k.\ P_{n+a_i}(f_i)) \Rightarrow P_n(\texttt{mkSubstFunc}_n(o, k, a, f))
\end{aligned}$$

We need to show:

$$\forall n : \mathbb{N}, f : \mathsf{SubstFunc}_n.\ P_n(f)$$

This is proved by (strong) induction on the size of the substitution function, as defined above. The goal is turned into an equivalent form:

$$\begin{aligned}
&\forall s : \mathbb{N}.\\
&\quad \forall n : \mathbb{N}, f : \mathsf{SubstFunc}_n.\ \texttt{sizeof}_n(f) = s \Rightarrow P_n(f)
\end{aligned}$$

To prove this, we assume that this is true for all values of s smaller than c, which is some integer:

$$\forall n : \mathbb{N}, f : \mathsf{SubstFunc}_n.\ \texttt{sizeof}_n(f) < c \Rightarrow P_n(f) \tag{5.8}$$

and the goal is:

$$\forall n : \mathbb{N}, f : \mathsf{SubstFunc}_n.\ \texttt{sizeof}_n(f) = c \Rightarrow P_n(f)$$

so let n, f, and c be such that $\texttt{sizeof}_n(f) = c$, we need to show: $P_n(f)$.

There are two cases. In the first case, $c = 0$ — according to Corollary 5.2.19 (Zero size is projection), this implies that:

$$\exists i : 1...n.\ f = \pi_n^i$$

which is true by (5.6). In the second case $c > 0$, and using Corollary 5.2.20 (Positive size is `mkSubstFunc`), this means that

$$\begin{aligned}&\exists o : \mathsf{OpId}, k : \mathbb{N}, a : (1...k \to \mathbb{N}), g : ((i : 1...k) \to \mathsf{SubstFunc}_{n+a_i}.\\ &\quad f = \texttt{mkSubstFunc}_n(o, k, a, g)\\ &\quad \&\ \texttt{sizeof}_n(f) = 1 + \sigma_{i=1}^{k}\, \texttt{sizeof}_{n+a_i}(g_i)\end{aligned}$$

which means that the size of f is bigger than each of the gs:

$$\begin{aligned}&\exists o : \mathsf{OpId}, k : \mathbb{N}, a : (1...k \to \mathbb{N}), g : ((i : 1...k) \to \mathsf{SubstFunc}_{n+a_i}.\\ &\quad f = \texttt{mkSubstFunc}_n(o, k, a, g)\\ &\quad \&\ \forall i : 1...k.\ \texttt{sizeof}_{n+a_i}(g_i) < \texttt{sizeof}_n(f) = c\end{aligned}$$

and according to (5.8) we get:

$$\begin{aligned}&\exists o : \mathsf{OpId}, k : \mathbb{N}, a : (1...k \to \mathbb{N}), g : ((i : 1...k) \to \mathsf{SubstFunc}_{n+a_i}.\\ &\quad (\forall i : 1...k.P_{n+a_i}(g_i))\\ &\quad \&\ f = \texttt{mkSubstFunc}_n(o, k, a, g)\end{aligned}$$

now, using (5.7) we get:

$$P_n(\texttt{mkSubstFunc}_n(o, k, a, g))$$

or:

$$P_n(f)$$

QED.

5.2.22 Theorem 22: `SubstFunc` recursion

Using Theorem 5.2.21 (`SubstFunc` induction), we can justify definitions like:

$$\begin{aligned}&R_n(\pi_n^i) = b(n, i)\\ &R_n(\texttt{mkSubstFunc}_n(o, k, a, f)) = c(n, o, k, a, f, \lambda i.\ r_{n+a_i}(f_i))\end{aligned}$$

Proving this would be standard, but very verbose, so a proof is not given here. This would be a more full version of the above claim:

$$
\begin{array}{l}
\forall T : \mathbb{U}. \\
\forall b : (n : \mathbb{N} \to 1...n \to T). \\
\forall c : (n : \mathbb{N} \to \mathsf{OpId} \to k : \mathbb{N} \to a : (1...k \to \mathbb{N}) \\
\quad \to ((i : 1...k) \to \mathsf{SubstFunc}_{n+a_i}) \to (1...k \to T) \to T). \\
\quad \exists! R : (n : \mathbb{N} \to \mathsf{SubstFunc}_n \to T). \\
\qquad \forall n : \mathbb{N}, i : 1...n. \\
\qquad\quad R_n(\pi_n^i) = b(n, i) \\
\qquad \forall n : \mathbb{N}, o : \mathsf{OpId}, k : \mathbb{N}, a : (1...k \to \mathbb{N}), f : ((i : 1...k) \to \mathsf{SubstFunc}_{n+a_i}). \\
\qquad\quad R_n(\mathtt{mkSubstFunc}_n(o, k, a, f)) = c(n, o, k, a, f, \lambda i.\ R_{n+a_i}(f_i))
\end{array}
$$

when a fixed T output type is all that is needed (which is enough for us). When T is parameterized by the inputs, this gets even more verbose:

$$
\begin{array}{l}
\forall T : (n : \mathbb{N} \to \mathsf{SubstFunc}_n \to Univ). \\
\forall b : (n : \mathbb{N} \to 1...n \to (n, \pi_n^i)). \\
\forall c : (n : \mathbb{N} \to o : \mathsf{OpId} \to k : \mathbb{N} \to a : (1...k \to \mathbb{N}) \\
\quad \to f : ((i : 1...k) \to \mathsf{SubstFunc}_{n+a_i}) \\
\quad \to (i : 1...k \to T(n + a_i, f_i)) \\
\quad \to T(n, \mathtt{mkSubstFunc}_n(o, k, a, f))). \\
\quad \exists! R : (n : \mathbb{N} \to f : \mathsf{SubstFunc}_n \to T(n.f)). \\
\qquad \forall n : \mathbb{N}, i : 1...n. \\
\qquad\quad R_n(\pi_n^i) = b(n, i) \\
\qquad \forall n : \mathbb{N}, o : \mathsf{OpId}, k : \mathbb{N}, a : (1...k \to \mathbb{N}), f : ((i : 1...k) \to \mathsf{SubstFunc}_{n+a_i}). \\
\qquad\quad R_n(\mathtt{mkSubstFunc}_n(o, k, a, f)) = c(n, o, k, a, f, \lambda i.\ R_{n+a_i}(f_i))
\end{array}
$$

5.2.23 Definition 23: `q` — quotation function

Defining a quotation function based on Definition 5.2.13 (`mkSubstFunc`) and Theorem 5.2.22 (`SubstFunc` recursion) is particularly simple. Usually we would recurse down the term keeping track of bindings that should not be quoted. In Section 3.5.4 the method that was used to implement '`rquote`' is to scan down a term structure, shifting operators up, while keeping track of bound variable names (using Nuprl's '`sweep_up_map_with_bvars`') so bound occurrences are not shifted. Using '`mkSubstFunc`' makes this simpler because its recursion never "loses bindings" until it reaches a base-case projection function. In fact, the MetaPRL [37, 36] reflection facility [55], which is based on our work, uses substitution functions as the

basic building blocks for *bound* terms, through the '`bterm`' operator. Using bound terms instead of terms as the basic type makes things a little easier, especially in the context of MetaPRL which uses this approach in its implementation.

'$\mathtt{q}$' is therefore defined first on substitution functions:

$$\forall n : \mathbb{N}, i : 1...n.\, \mathtt{q}_n(\pi_n^i) \equiv \pi_n^i$$
$$\forall n : \mathbb{N}, o : \mathsf{OpId}, k : \mathbb{N}, a : (1...k \to \mathbb{N}), f : ((i : 1...k) \to \mathsf{SubstFunc}_{n+a_i}).$$
$$\mathtt{q}_n(\mathtt{mkSubstFunc}_n(o, k, a, f)) \equiv \mathtt{mkSubstFunc}_n(\underline{o}, k, a, \lambda i.\ \mathtt{q}_{n+a_i}(f_i))$$

Finally, quotation of terms is defined using the one for substitution functions and the relation between '`mkTerm`' and '`mkSubstFunc`' mentioned above:

$$\forall t : \mathsf{Term}.\ \mathtt{q}(t) \equiv (\mathtt{q}_0(\lambda.\, t))(\langle\rangle)$$

where '$\langle\rangle$' denotes the empty tuple. Using this relation we get:

$$\begin{aligned}
&\forall o : \mathsf{OpId}, k : \mathbb{N}, a : (1...k \to \mathbb{N}), f : ((i : 1...k) \to \mathsf{SubstFunc}_{a_i}).\\
&\quad \mathtt{q}(\mathtt{mkTerm}(o, k, a, f))\\
&\quad = (\mathtt{q}_0(\lambda.\, \mathtt{mkTerm}(o, k, a, f)))(\langle\rangle)\\
&\quad = (\mathtt{q}_0(\mathtt{mkSubstFunc}_0(o, k, a, f)))(\langle\rangle)\\
&\quad = (\mathtt{mkSubstFunc}_0(\underline{o}, k, a, \lambda i.\ \mathtt{q}_{a_i}(f)))(\langle\rangle)\\
&\quad = (\lambda.\, \mathtt{mkTerm}(\underline{o}, k, a, \lambda i.\ \mathtt{q}_{a_i}(f)))(\langle\rangle)\\
&\quad = \mathtt{mkTerm}(\underline{o}, k, a, \lambda i.\ \mathtt{q}_{a_i}(f))
\end{aligned}$$

5.2.24 Theorem 24: `q` represents atoms

This is the first step in proving that '$\mathtt{q}$' is a valid representation function (*i.e.*, that it is good for requirements (d) and (e), pp. 84). We need to verify that '$\mathtt{q}$' works on atomic term:

$$\forall t : \mathsf{Atom}.\, \mathtt{q}(t)\ \mathtt{reps}\ t$$

So let o be some opid and t be an atom made with it:

$$t = \mathtt{mkTerm}(o, 0, \lambda i.\ ?, \lambda i.\ ?)$$

By Definition 5.2.23 ($\mathtt{q}$) definition:

$$\begin{aligned}
\mathtt{q}(t) &= \mathtt{q}(\mathtt{mkTerm}(o, 0, \lambda i.\ ?, \lambda i.\ ?))\\
&= \mathtt{mkTerm}(\underline{o}, 0, \lambda i.\ ?, \lambda i.\ ?)
\end{aligned}$$

and because terms with shifted opids are canonical:

$$\mathtt{mkTerm}(\underline{o}, 0, \lambda i.\ ?, \lambda i.\ ?)\ \underline{\mathtt{evalsto}_{\mathtt{o}}}\ \mathtt{mkTerm}(\underline{o}, 0, \lambda i.\ ?, \lambda i.\ ?)$$

According to this,

$$\mathtt{mkTerm}(\underline{o}, 0, \lambda i.\ ?, \lambda i.\ ?)\ \mathtt{reps}\ \mathtt{mkTerm}(o, 0, \lambda i.\ ?, \lambda i.\ ?)$$

is trivially true (the $\forall$ clause is trivially true since $k = 0$).

5.2.25 Theorem 25: q represents substitutions

The second step is demonstrating that '$\mathtt{q}$' of a substitution function produces a representation of it:

$$\forall n : \mathbb{N}, f : \mathsf{SubstFunc}_n, t : \mathsf{Atom}^n.\ \mathtt{q}_n(f)(\mathtt{map}(\mathtt{q}, t))\ \mathtt{reps}\ f(t)$$

We prove this using Theorem 5.2.21 (SubstFunc induction). The base case is when:

$$\exists i : 1...n.\ f = \pi_n^i$$

By Definition 5.2.23 ($\mathtt{q}$), we get:

$$\begin{aligned}&\mathtt{q}_n(f) = f\\ &\Rightarrow \mathtt{q}_n(f)(\mathtt{map}(\mathtt{q}, t)) = f(\mathtt{map}(\mathtt{q}, t)) = \mathtt{q}(t_i)\end{aligned}$$

so we need to show:

$$\mathtt{q}(t_i)\ \mathtt{reps}\ t_i$$

which is true by Theorem 5.2.24 ($\mathtt{q}$ represents atoms).

Now for the inductive case — we assume:

$$\begin{aligned}&\exists o : \mathsf{OpId}, k : \mathbb{N}, a : (1...k \to \mathbb{N}), g : ((i : 1...k) \to \mathsf{SubstFunc}_{n+a_i}).\\ &\quad \forall i : 1...k, r : \mathsf{Atom}^{n+a_i}.\ \mathtt{q}_{n+a_i}(g_i)(\mathtt{map}(\mathtt{q}, r))\ \mathtt{reps}\ g_i(r)\\ &\quad \&\ f = \mathtt{mkSubstFunc}_n(o, k, a, g)\end{aligned}$$

and our goal is:

$$\mathtt{q}_n(\mathtt{mkSubstFunc}_n(o, k, a, g))(\mathtt{map}(\mathtt{q}, t))\ \ \mathtt{reps}\ \mathtt{mkSubstFunc}_n(o, k, a, g)(t)$$

According to Definition 5.2.23 ($\mathtt{q}$) the goal becomes:

$$\mathtt{mkSubstFunc}_n(\underline{o}, k, a, \lambda i.\ \mathtt{q}_{n+a_i}(g_i))(\mathtt{map}(\mathtt{q}, t))\ \ \mathtt{reps}\ \mathtt{mkSubstFunc}_n(o, k, a, g)(t)$$

and using Definition 5.2.13 (mkSubstFunc) the goal becomes:

$$\begin{aligned}&\lambda t : \mathsf{Term}^n.\\ &\quad \mathtt{mkTerm}(\underline{o}, k, a, \lambda i : 1...k.\ \lambda s : \mathsf{Term}^{a_i}.\ \mathtt{q}_{n+a_i}(g_i)(\mathtt{append}(t, s)))(\mathtt{map}(\mathtt{q}, t))\\ &\mathtt{reps}\\ &\lambda t : \mathsf{Term}^n.\\ &\quad \mathtt{mkTerm}(o, k, a, \lambda i : 1...k.\ \lambda s : \mathsf{Term}^{a_i}.\ g_i(\mathtt{append}(t, s)))(t)\end{aligned}$$

The outermost function applications yield this goal:

$$\begin{aligned}&\mathtt{mkTerm}(\underline{o}, k, a, \lambda i : 1...k.\ \lambda s : \mathsf{Term}^{a_i}.\ \mathtt{q}_{n+a_i}(g_i)(\mathtt{append}(\mathtt{map}(\mathtt{q}, t), s)))\\ &\mathtt{reps}\\ &\mathtt{mkTerm}(o, k, a, \lambda i : 1...k.\ \lambda s : \mathsf{Term}^{a_i}.\ g_i(\mathtt{append}(t, s)))\end{aligned}$$

Now the left term is canonical since it has a shifted operator id, so to satisfy Definition 5.2.6 (reps), it is enough to show:

$$\begin{array}{l}\forall i : 1...k, u : \mathsf{Atom}^{a_i}. \\ \quad \mathtt{q}_{n+a_i}(g_i)(\mathtt{append}(\mathtt{map}(\mathtt{q}, t), \mathtt{map}(\mathtt{q}, u))) \\ \quad\quad \mathtt{reps}\ g_i(\mathtt{append}(t, u))\end{array}$$

Let r be (under the above qualifiers):

$$r : \mathsf{Atom}^{n_{a_i}}.\ r = \mathtt{append}(t, u)$$

and using facts on composing 'append' and 'map' the goal can be written as:

$$\mathtt{q}_{n+a_i}(g_i)(\mathtt{map}(\mathtt{q}, r))\ \mathtt{reps}\ g_i(r)$$

which is exactly our induction hypothesis.
QED.

5.2.26 Theorem 26: q is a representation (requirements (d) & (e))

$$\forall t : \mathsf{Term}.\ \mathtt{q}(t)\ \mathtt{reps}\ t$$

Proving this is easy using Theorem 5.2.25 (q represents substitutions). First, write t explicitly:

$$\begin{array}{l}\exists o : \mathsf{OpId}, k : \mathbb{N}, a : (1...k \to \mathbb{N}), f : ((i : 1...k) \to \mathsf{SubstFunc}_{a_i}). \\ \quad t = \mathtt{mkTerm}(o, k, a, f)\end{array}$$

and we need to show:

$$\mathtt{q}(\mathtt{mkTerm}(o, k, a, f))\ \mathtt{reps}\ \mathtt{mkTerm}(o, k, a, f)$$

Using Definition 5.2.23 (q) the goal is:

$$\mathtt{mkTerm}(\underline{o}, k, a, \lambda i.\ \mathtt{q}_{a_i}(f))\ \mathtt{reps}\ \mathtt{mkTerm}(o, k, a, f)$$

Again, since the term on the left is canonical, it is enough to show:

$$\forall i : 1...k, t : \mathsf{Atom}^{a_i}.\ \mathtt{q}_{a_i}(f)(\mathtt{map}(\mathtt{q}, t))\ \mathtt{reps}\ f_i(t)$$

which is true by Theorem 5.2.25 (q represents substitutions).

5.2.27 Theorem 27: Upward HOAS

This is an important theorem that claims that when some substitution functions represent other substitution functions, then plugging them into appropriate terms makes one represent the other as well. Note that it is desirable to have the reverse

of this, but it seems like it is impossible (basically, the "behavioral requirement" placed on substitution functions can hold only on term representations). Another note is that elegant formulations of '$\texttt{reps}$' (*e.g.*, '$A \,\texttt{reps}\, B$' defined as '$A \cong \texttt{q}(B)$') have failed satisfying this property.

$$\begin{array}{l}\forall o : \mathsf{OpId}, k : \mathbb{N}, a : (1...k \to \mathbb{N}), f, g : ((i : 1...k) \to \mathsf{SubstFunc}_{a_i}). \\ \quad (\forall i : 1...k, t, s : \mathsf{Term}^{a_i}. \\ \qquad (\forall j : 1...a_i . t_j \,\texttt{reps}\, s_j) \\ \qquad \Rightarrow g_i(t) \,\texttt{reps}\, f_i(s)) \\ \quad \Rightarrow \texttt{mkTerm}(\underline{o}, k, a, g) \;\; \texttt{reps} \;\; \texttt{mkTerm}(o, k, a, f)\end{array} \tag{5.9}$$

Assume all variables given as specified, our goal is:

$$\texttt{mkTerm}(\underline{o}, k, a, g) \;\; \texttt{reps} \;\; \texttt{mkTerm}(o, k, a, f)$$

Since the left term is canonical, then by Definition 5.2.6 (`reps`) it enough to show:

$$\forall i : 1...k, r : \mathsf{Atom}^{a_i}. \; g_i(\texttt{map}(\texttt{q}, r)) \;\; \texttt{reps} \;\; f_i(r)$$

Assume such i and r are given, an instantiation of (5.9) gives us:

$$\begin{array}{l}(\forall j : 1...a_i . \texttt{map}(\texttt{q}, r)_j \,\texttt{reps}\, r_j) \\ \Rightarrow g_i(\texttt{map}(\texttt{q}, r)) \,\texttt{reps}\, f_i(r)\end{array}$$

which is:

$$\begin{array}{l}(\forall j : 1...a_i . \texttt{q}(r_j) \,\texttt{reps}\, r_j) \\ \Rightarrow g_i(\texttt{map}(\texttt{q}, r)) \,\texttt{reps}\, f_i(r)\end{array}$$

but the antecedent is true by Theorem 5.2.26 (`q` is a representation) (actually, Theorem 5.2.24 (`q` represents atoms) is sufficient), so we know:

$$g_i(\texttt{map}(\texttt{q}, r)) \,\texttt{reps}\, f_i(r)$$

which is what we needed to show.
QED.

5.2.28 Definition 28: `unq` — unquotation function

An unquote function is defined in the same way as Definition 5.2.23 (`q`), except that it unquotes its input. This is similar to the fact that '`runquote`' mirrors '`rquote`' (Section 3.5.4), doing the same kind of '`sweep_up_map_with_bvars`' scan.

The '`unq`' definition is therefore defined on substitution functions first:

$$\begin{array}{l}\forall n : \mathbb{N}, i : 1...n. \, \texttt{unq}_n(\pi^i_n) \equiv \pi^i_n \\ \forall n : \mathbb{N}, o : \mathsf{OpId}, k : \mathbb{N}, a : (1...k \to \mathbb{N}), f : ((i : 1...k) \to \mathsf{SubstFunc}_{n+a_i}). \\ \quad \texttt{unq}_n(\texttt{mkSubstFunc}_n(\underline{o}, k, a, f)) \equiv \texttt{mkSubstFunc}_n(o, k, a, \lambda i. \;\; \texttt{unq}_{n+a_i}(f_i))\end{array}$$

(Note that with projection functions both quotation and unquotation are simply an identity). This is then used to define unquotation of Terms:

$$\forall t : \mathsf{Term}.\ \mathtt{unq}(t) \equiv (\mathtt{unq}_0(\lambda.\ t))(\langle\rangle)$$

And just like the '$\mathtt{q}$' case, we get:

$$\begin{aligned}
&\forall o : \mathsf{OpId}, k : \mathbb{N}, a : (1...k \to \mathbb{N}), f : ((i : 1...k) \to \mathsf{SubstFunc}_{a_i}).\\
&\quad \mathtt{unq}(\mathtt{mkTerm}(\underline{o}, k, a, f))\\
&\quad = (\mathtt{unq}_0(\lambda.\ \mathtt{mkTerm}(\underline{o}, k, a, f)))(\langle\rangle)\\
&\quad = (\mathtt{unq}_0(\mathtt{mkSubstFunc}_0(\underline{o}, k, a, f)))(\langle\rangle)\\
&\quad = (\mathtt{mkSubstFunc}_0(o, k, a, \lambda i.\ \mathtt{unq}_{a_i}(f)))(\langle\rangle)\\
&\quad = (\lambda.\ \mathtt{mkTerm}(o, k, a, \lambda i.\ \mathtt{unq}_{a_i}(f)))(\langle\rangle)\\
&\quad = \mathtt{mkTerm}(o, k, a, \lambda i.\ \mathtt{unq}_{a_i}(f))
\end{aligned}$$

5.2.29 Theorem 29: unq inverse of q

It is easy to see that '$\mathtt{unq}$' is the inverse of '$\mathtt{q}$':

$$\forall t : \mathsf{Term}.\ \mathtt{unq}(\mathtt{q}(t)) = t. \tag{5.10}$$

Note, however, that the other side is not always true:

$$\neg\forall t : \mathsf{Term}.\ \mathtt{q}(\mathtt{unq}(t)) = t,$$

because '$\mathtt{unq}$' is meaningless on terms that are not quoted. If we want both sides to be true, then we must ensure that t is a representation of some term:

$$\forall t : \mathsf{Term}.\ (\exists t' : \mathsf{Term}.\ t = \mathtt{q}(t')) \ \Rightarrow\ \mathtt{unq}(\mathtt{q}(t)) = t = \mathtt{q}(\mathtt{unq}(t)).$$

It is simple to verify that this holds using (5.10):

$$\begin{aligned}
\mathtt{q}(\mathtt{unq}(t)) &= \mathtt{q}(\mathtt{unq}(\mathtt{q}(t')))\\
&= \mathtt{q}(t') && (using (5.10))\\
&= t
\end{aligned}$$

Chapter 6
Applying Reflection

6.1 Motivating Example

At long last, we get to apply our reflection scheme in Nuprl, using it to accomplish a syntactic (meta-mathematical) proof. The proof that is used to motivate this is Tarski's result regarding the undefinability of truth inside a logic. We begin with a standard proof that is written in a conventional concrete style[1]. This original version appears as a 'comment object' at the beginning of the '`tarski`' theory, which can be found in Appendix B.2.

A pleasing (and unexpected) result is that we could accomplish this proof in Nuprl by following the exact same steps that were used in the manual version. Using the system enables the usual benefits: the proof is formal, providing a rigorous verification of the original manual version; the proof itself involves a few complicated and confusing steps that are much easier to deal with when the system accounts for all details; and it can be used as a pedagogic tool to understand both this specific proof and the general techniques that are involved in syntactical and meta-mathematical formal content. To enhance the connection between the manual and the automatic versions, we use the display form techniques that were discussed in Section 3.5.3, converting the manual proof text (the comment object) into a hypertext document with clickable links from it to corresponding parts of the Nuprl proof. This makes for a good high-level overview that enhances the user-interaction experience. This technique can be used anywhere in Nuprl, especially when the color code is incorporated into version 5 of the system that has enhanced interactive features.

We now continue with an overview of various parts of the '`reflection`' and the '`tarski`' theory files. This overview will expose a few important concerns, and the relevant discussions will be intermixed. Note that '`reflection`' contains some rules that are not needed for '`tarski`', but are provided for other usages. The descriptions will usually be brief high-level summaries, focusing on points that need an explanation. Appendix B contains a full printout of these theories which can be consulted for a full content listings, exact definition, and theorem proofs. The presentation involves extensive use of the ellipsis syntactic notation, so we begin by a brief overview of this facility.

6.2 Extended Syntactic Notation

The notation used in many of the rules depends on varying the numbers of bindings and subterms. For this, we continue using the Scheme-like ellipsis notation that was

[1]This proof was written in February 2001 by Stuart Allen. It is interesting to note that it was written as an example of what we thought would not be possible due to its usage of a quoted (concrete) free variable name.

mentioned in Section 3.1. In Scheme, the ellipsis notation is used to specify hygienic syntax transformation (*e.g.*, in '`syntax-rules`') rules, so it is fully specified and unambiguous: it is possible to convert these patterns to explicit notations using index functions that were used in the previous chapters. Given that this notation has an exact specification in Scheme, the following description is an informal brief overview, with a few simple extensions.

It should be mentioned that a similar pattern matching facility *could* be incorporated into Nuprl's rule specification. Currently, the system provides rather limited functionality, due to large amounts of existing legacy code combined with the fact that there was no stressing need for such functionality so far. As a result, many additional rules are implemented in Lisp code rather than being specified as theory objects. A richer system is required when dealing with a general logical framework, which is why the MetaPRL implementation [36] has extensive support for rich rewrite rule specifications.

The ellipsis notation is used in contexts that have a sequence of syntactical elements. Such contexts include bindings, subterms, subgoals, and rule arguments. In a Nuprl rule specification, 'pattern variables' are used as template holes, for example, in the rule:

$H \vdash$ `foo(`a`)`
by `FooRule` b
* $H \vdash$ `bar(`b`)`
* H, `bar(`b`)` $\vdash$ `foo(`a`)`

both 'a ' and 'b ' are pattern variables, instantiated from the goal which the rule is applied on, and the argument given to the rule. The ellipsis notation that we add is a '...' token, which can only be used in a syntactical context that has a sequence of sub-syntaxes — never as the first element, and usually as the last one. Whenever '...' is used in a pattern, it means that the previous pattern is allowed to repeat zero or more times, and all pattern variables that appear in the previous sub-syntax template are bound to a sequence of the corresponding subparts of the sequence of matched syntax elements. Usages of '...' can be nested to match sequences of sequences. This will be clearer given a few examples — the following should be enough to understand how this work in all cases (note that all of these are patterns on a left hand side of a rule):

'`foo(`a`;...)`' Matches all terms that have a '`foo`' operator id with any number of subterms, and 'a' matches this sequence of subterm syntaxes. For example, when matched against '`foo(1;2;3)`', 'a' matches the sequence '`1`', '`2`', '`3`', and when matched against '`foo()`', 'a' matches the empty sequence.

'`foo(`a`;`b`;...)`' This matches terms similar to the above, except that the '`foo`' term should have at least one subterm. The first subterm is matched to 'a', and the rest matches 'b' (as a sequence).

'`foo(0;0;...)`' This matches terms with a '`foo`' opid and one or more subterms that are all '`0`'.

‘`foo(bar(`a`); ...)`’ This matches terms that have a ‘`foo`’ operator id, and zero or more subterms, each of which has a ‘`bar`’ operator id and a single subterm. ‘a’ matches the sequence of the single subterms of the ‘`bar`’ subterms. For example, when matched against ‘`foo(bar(1); bar(2))`’, ‘a’ matches the sequence ‘`1`’, ‘`2`’.

‘`foo(bar(`a`;`b`); ...)`’ Similar to the above, except that all ‘`bar`’ subterms have two subterms each, with ‘a’ bound to the sequence of all first subterms and ‘b’ to all seconds. For example, when matched against ‘`foo(bar(1;2); bar(3;4))`’, ‘a’ matches the sequence ‘`1`’, ‘`3`’, and ‘b’ matches ‘`2`’, ‘`4`’.

‘`foo(bar(`a`;...); ...)`’ Similar to the above, except that all ‘`bar`’ subterms have zero or more subterms, with ‘a’ bound to the sequence of sequences of these subterms. For example, when matched against ‘`foo(bar(1); bar(2;3))`’, ‘a’ matches the sequence containing the sequence ‘`1`’ and the sequence ‘`2`’, ‘`3`’.

‘`foo(bar(`a`;`b`;...); ...)`’ Similar to the above, except that all ‘`bar`’ subterms have one or more subterms, ‘a’ matches the sequence of all first subterms, and ‘b’ matches the sequence of the sequences of the rest in all subterms.

‘`foo(bar(a;`a`); ...)`’ The thing to note here is that if we want to match ‘`bar`’ terms with two subterms, the first is an ‘`a`’ term and the second is some an arbitrary term then we need some way to distinguish the two. In this text we use a different font for the two as demonstrated by this template, which binds ‘a’ to the sequence of the second subterms of the ‘`bar`’ subterms.

When these pattern variables are used on a rule's right hand side, they should appear in exactly the same kind sequence context, including the same nesting level. This produces a result where each such sequence variable plugs in its values in a matching number of syntactic elements. For example, the following rule:

$H \vdash$ `foo(bar(`a`;`b`;...); ...)`
  by `Blah`
  * $H \vdash$ `bar(`a`;`b`;...)`
  ...

matches on a goal term as specified, and generates a subgoal for every ‘`bar`’ subterm. In addition, sequence variables can be used in contexts that restricts them to match the same number of elements. For example, ‘`foo(bar(a;...); baz(b;...))`’ matches ‘`foo`’ terms with a ‘`bar`’ and ‘`baz`’ subterms each containing any number of subterms, but when used in this rule:

$H \vdash$ `foo(bar(a;...); baz(b;...))`
  by `Blah`
  * $H \vdash$ `foo(`a`;`b`)`
  ...

then a restriction is added so the rule matches only instances where the number of ‘`bar`’ subterms is equal to the number of ‘`baz`’ subterms. (This is an extension that is not part of the Scheme usage of ellipsis.)

6.3 The 'reflection' theory

6.3.1 term part

The 'reflection' theory begins with a definition of a 'term' type which will stand for our abstract terms. Its display form is set to show as 'Term', avoiding the confusion with the built-in ML-level 'term' type which is the system's type for concrete terms. Several standard items are included to make this a valid type:

- 'termFormation' and 'termEquality' rules: these are required for all defined types. The rules are added to the 'Trivial' tactic to make the system solve such goals automatically.
- 'term_wf' is a simple theorem stating that the new type is well-formed: Term $\in \mathbb{U}_i$. It is trivially proved with the previous rules.
- 'init_term' is an ML code object which adds the above rules to the 'Trivial' tactic.

6.3.2 is_subst part

'is_subst' is the term that is used as a predicate goal for identifying substitution functions through a syntactical analysis. As previously described, it is used with an arbitrary number of binders:

is_subst(v,....e).

However, Nuprl has no way for defining a display form (and aliases for the structure editor) that involve a varying number of binders, so this section begins by defining such display forms for 'is_subst$_0$' through 'is_subst$_8$' for most cases.

is_substFormation rule

This is the first rule that demonstrates the limitation of Nuprl's rule facilities. The rule should handle formation of an 'is_subst' type together with its extract — but this involves all operators with an id of 'is_subst', and a single subterm of *arbitrary* arity. Because of this, the rule is implemented using two ML functions:

- 'is_substFormation_rule' — for the rule;
- 'is_substFormation_extract_maker' — for creating extracts.

Using our extended ellipsis notation, this rule is simply:

$H \vdash$ $\mathbb{U}_i$ ext is_subst(v,....e) : Term
  by is_substFormation [v; ...]
  * H, v : Term, ... $\vdash$ Term ext e
  ...

`is_substEquality` rule

For equality of two '`is_subst`' instances, done through the ML function '`is_substEquality_rule`' due to varied arity:

$H \vdash$ `is_subst(`x`,....`a`)` = `is_subst(`y`,....`b`)` $\in \mathbb{U}_i$
  by `is_substEquality` $[v; ...]$
  * $H, v :$ `Term`$, ... \vdash\ a[v/x, ...] = b[v/y, ...] \in$ `Term`

Note that the substitution notation here looks like the concrete substitution notation that was introduced in Chapter 4, but it is, in fact, an abstract substitution since the concrete syntax is not available in Nuprl's object level language.

6.3.3 `term_eq` part

This contains some definitions, rules, and theorems that are expected from new Nuprl types. This includes:

- '`term_eq`' — an equality decision for '`Term`'s. '`term_eq(a;b;s;t)`' evaluates '`a`' and '`b`', and results in '`s`' or '`t`' depending on whether they are equal or not.
- '`term_eqEquality`' rule — used for '`term_eq`' terms, and used by a tactic by the same name that is added to the '`EqCD`' tactic.
- '`term_eqReduceTrue`' and '`term_eqReduceFalse`' — two rules for reducing a true or a false outcome of '`term_eq`'.
- '`eq_term`' — a boolean-valued predicate that uses '`term_eq`'[2]. Defined as `eq_term(x;y)` ≡ `term_eq(x;y;tt;ff)`.
- '`decidable__term_equal`' — a theorem that states that '`Term`' equality is decidable. (The format of this name is required by the system.)

6.3.4 `TermAuto` part

This part contains the important term well-formedness rules that were discussed in Chapter 4.

`isTerm` rule

This is the simple top-level rule that turns a '`Term`' membership (and α-equality) goal into an '`is_subst`' goal (as discussed in Section 4.6.1).

$H \vdash\ t = t \in$ `Term`
  by `isTerm`
  * $H\ \vdash$ `is_subst(.`t`)`

[2]This confusing naming scheme is used elsewhere in Nuprl.

An important thing to note here is that Nuprl uses α-equality to verify that such a rule can be used, therefore this rule can be used with any two α-equivalent terms: at the object language level, two α-equivalent terms are indistinguishable, and this is not an exception. A whole class of problems therefore disappears in one fell swoop.

`isSubstTerm` rule

This rule turns a simple instance of '`is_subst`' with no bindings back into a '`Term`' membership goal:

$H \vdash$ `is_subst(.`t`)`
by `isSubstTerm`
* $H \vdash t \in$ `Term`

The reason that this is needed is closely related to the fact that we can use descriptions that are not fully quoted literal terms: to prove that such a description is a term, we will need to use additional facts from the hypothesis list. For example, proving that '`λx.x + y`' is a term, when we know that $y \in$ Term. See '`isSubstThin`' below for additional details.

`isSubst` rule

This is main rule from Section 4.6.1 that scans down a term structure (actually, an '`is_subst`' goal), making sure it has the right form. It is implemented using the ML function '`is_subst_rule`' due to varied arity of the '`is_subst`' goal. This rule has two possible templates, first, for the simple projection case:

$H \vdash$ `is_subst(`x`,...,`v`,`y`,....`v`)`
by `isSubst`
(no subgoals)

and another, for a shifted operator id of the subterm:

$H \vdash$ `is_subst(`x`,....op(`y`,...`b`;...)`
by `isSubst`
* $H \vdash$ `is_subst(`x`,...,`y`,....`b`)`
...

Note that this rule will succeed only for terms that are either fully quoted or can be rewritten to such. In some cases we need to use '`isSubstTerm`' above to escape back to where we can use existing hypotheses, and in some examples such as:

$\vdash$ `is_subst(x,y.if p(x) then y else y)`

we will have to show that the '`if`' subterm can be rewritten to just a simple '`y`', and use that to perform the rewrite in the goal.

`isTermSubst` rule

This rule is used with term equality goals, where the two sides of the equality are not α-equal, but can be proved as such. It produces a '`Term`' membership subgoal and equality subgoals for each subterm, implemented by the ML '`isTermSubst_rule`' function.

$H \vdash$ `op(`x_1`,...,`b_1`;...)` $=$ `op(`x_2`,...,`b_2`;...)` $\in$ `Term`
by `isTermSubst` $[v; ...; ...]$
* $H \vdash$ `op(`x_1`,...,`b_1`;...)` $\in$ `Term`
* H, v : `Term`, ... $\vdash$ $b_1[v/x_1, ...] = b_2[v/x_2, ...] \in$ `Term`
...

`isSubstThin` rule

A rule for thinning out the given variables, provided they are not used in the original '`is_subst`' goal. Implemented by the '`is_substThin_rule`' ML function.

$H \vdash$ `is_subst(`x`,....`b`)`
by `isSubstThin` $[v; ...]$
* $H \vdash$ `is_subst(`y`,....`b`)`
where $v, ...$ are all from $x, ...$ and none of them are free in b,
and $y, ...$ are the same as $x, ...$ after removing $v, ...$

There is also a tactic for this rule, '`IsSubstThin`', which automatically chooses all possible variables for thinning. Note that proving this rule requires term induction.

This rule is important when used with '`isSubstTerm`': the fact is that we cannot convert any '`is_subst`' goals to term membership — doing this will re-introduce the problem of exotic terms. We *can* do this, however, with '`is_subst`' goals that have no bound variables. This way, we cannot state any facts on bound variables, so proving '`is_subst`$_1$`(x.x)`' must be done without any help of some hypothesis. Therefore, '`isSubstThin`' makes it possible to use '`isSubstTerm`' properly: in places where we need to use a hypothesis, we must show that the term in question has no bindings. For example: when proving

$$y : \texttt{Term} \vdash \texttt{\underline{\lambda x . x + y}}$$

we will end up with two subgoals:

- y : `Term` $\vdash$ `is_subst(x.x)`, which is easily proved using the projection function base case of '`isSubst`', and
- y : `Term` $\vdash$ `is_subst(x.y)`, which requires thinning '`x`' out and converting the nullary '`is_subst`' back to term membership using '`isSubstTerm`' before the hypothesis can be used.

`TermAuto` tactic

This is a "super" tactic that uses '`isTerm`' to convert a membership or an equality goal into an '`is_subst`' goal (and uses '`isTermSubst`' for non-trivial equalities). If the goal is an '`is_subst`' goal it will try to use existing assumptions that prove the goal (using '`isSubstTerm`'), otherwise, it will continue with an '`isSubst`' step and thinning out all unused variables.

This tactic is added to the '`Auto`' tactic, which makes most easy '`Term`' goals easy to solve.

6.3.5 `termin/termof` part

`termin` definition

'`termin`' is defined as a new predicate type. The intention is to use to know when a given term represents a term in some type. A simple definition of such a relation can be might seem to be:

$$\forall a : \mathsf{Term}, A : \mathbb{U}_i.\ \ a\ \texttt{termin}\ A\ \equiv\ \texttt{unq}(a) \in A$$

but this cannot work in Nuprl's type theory, since type membership is used for well-formedness rather than a predicate. This is the reason for an explicit predicate that combines unquotation with type membership; this construction is very similar to the bar types of Constable and Crary [19, 20].

This part contains only the basic definitions that make '`termin`' into a new type: a '`terminEquality`' rule, a '`terminFormation`' rule, and a '`termin_wf`' theorem. The meaning of the predicate is established later through '`termin_member`'; it uses the '`down`' operation that is defined in the next part. The display form of '`termin`' is set to

$$a \downarrow\!\in A$$

according to the above intuition.

Using '`termin`', a new type constructor is defined: '`termof`'. Given a type T, `termof` T is the type of all terms that stand for terms of type T. It is easy to define this constructor in Nuprl as a simple abstraction:

$$\texttt{Termof}\ A\ \equiv\ \{t : \texttt{Term} \mid t \downarrow\!\in A\}$$

6.3.6 `up/down` part

As discussed in Chapter 3, our reflection implementation contains functions for concrete quotation and unquotation. These functions are part of the concrete level of the system, to be used only by the editor and similar low-level subsystems.

At the object language level, we need to define a quotation and an unquotation functions that correspond to the '$\texttt{q}(\cdot)$' and '$\texttt{unq}(\cdot)$' functions of Chapter 5. The problem is that these operations, as their concrete counterparts, operate on

complete terms — so exposing them to the object language level will mean that it will be inherently restricted to literal terms. This would not provide access to the full power of shifted operators which were designed to allow intermixing of quoted term parts with *term descriptions*.

The solution for this is therefore exposing user-level variations of '$\mathtt{q}(\cdot)$' and '$\mathtt{unq}(\cdot)$' which we will call '`up`' and '`down`'[3] and display as '$\uparrow\cdot$' and '$\downarrow\cdot$'. The versions are supposed to do their work in a step-by-step mode rather than operating on complete terms. In other words, we need to start with the algorithm that is implements these functions, and turn it into an algorithm that can works its way through a sequence of terms, where each step can be used to correctly continue the sequence. For example, if a Nuprl user has a term '`unq(`$\underline{1+2}$`)`', it is wrong to provide a simple computation hook for '`unq`' that evaluates to the unquotation of the addition — this will work with this example, but will fail if the quoted term contains a description, such as '$\underline{1}+x$'.

Except for the evaluation rules for '`up`' and '`down`', this part contains some standard definitions for the new term types: '`upFormation`', '`upEquality`', '`downFormation`', '`downEquality`', etc. The '`up`' related rules differ from the ones for '`down`' — '`up`' is used as an operation from '`Term`'s to '`Term`'s, while '`down`' goes from '`Termof A`' to '`A`'. The reason for the restriction on '`up`' is that we cannot use it with all inputs — there are some input values that we cannot find a representation for, for example, diverging terms have no value; other inputs have no easy way for finding a representation, like functions where the problem of finding a representation is the same as the problem of finding a body for an arbitrary function (which is a subject of research that is outside the scope of this text [10, 2]).

The main work is implemented by an ML function for performing the necessary steps: '`compute_up_down`' in '`reflection.ml`'. This function is then hooked into Nuprl's evaluation mechanism using '`set_compute_hook`'. The question is — how can we implement step-by-step versions of these functions, when '`q`' and '`unq`' work by a recursive term scan that keeps track of bound variables? If we were to naively push these operations inside their terms (after performing the operator shifting in the right direction), for example, saying that:

$$\downarrow\underline{\mathtt{foo(x.bar(x;2))}}\ \underline{\mathtt{evalsto}}\ \underline{\mathtt{foo(x.}\downarrow\mathtt{bar(x;2))}},$$

then we would lose information about bindings that we went through.

One possible solution is to turn these operations from ones that expect a simple subterm argument to operations that expect a bound subterm, where the bindings indicate variable names that should not be modified. Using this strategy, the

[3]The arbitrariness of direction can be a little confusing: we talk about 'quotation levels' (quotedness), so '`up`' increases this level and '`down`' decreases it.

computation for the above example would proceed along these steps:

$$\begin{aligned}\downarrow\underline{\texttt{foo(x.bar(x;2))}} &\ \underrightarrow{\texttt{evalsto}}\ \texttt{foo(x.}\downarrow\texttt{(x.}\underline{\texttt{bar(x;2)}}\texttt{))}\\ &\ \underrightarrow{\texttt{evalsto}}\ \texttt{foo(x.bar(}\downarrow\texttt{(x.x); }\downarrow\texttt{(x.}\underline{\texttt{2}}\texttt{)))}\\ &\ \underrightarrow{\texttt{evalsto}}\ \texttt{foo(x.bar(x;2))}\end{aligned}$$

It seems at first like this works as expected, but a problem pops up right after the first step: the '`x`' variable term is bound by the '↓' subterm rather than '`foo`'s subterm. The binding structure is wrong — for example, renaming the outermost '`x`' will not rename the bound occurrence too. This is evident in the fact that '`↓(x.x)`' is expected to evaluate to '`x`', making it invalid.

Theorem 5.2.29 (`unq` inverse of `q`) from Chapter 5 provides a hint for a better solution: if '`unq`' and '`q`' cancel each other out, then they can be used to wrap bound variable subterms in a 'layer of protection'. This leads to the following evaluation fragments which are implemented by '`compute_up_down`':

$$\downarrow\uparrow t\ \underrightarrow{\texttt{evalsto}}\ t \tag{6.1}$$

$$\downarrow\underline{\texttt{op(}x.b\texttt{)}}\ \underrightarrow{\texttt{evalsto}}\ \texttt{op(}x.\downarrow(b[\uparrow x/x])\texttt{)} \tag{6.2}$$

$$\uparrow\downarrow t\ \underrightarrow{\texttt{evalsto}}\ t \tag{6.3}$$

$$\uparrow\texttt{op(}x.b\texttt{)}\ \underrightarrow{\texttt{evalsto}}\ \underline{\texttt{op(}x.}\uparrow(b[\downarrow x/x])\underline{\texttt{)}} \qquad \text{(where 'op' is canonical)} \tag{6.4}$$

This evaluation works correctly, and allows free mixing of quoted terms and descriptions. A few things to note here are:

- Another intuition that demonstrates why these rules work is that '↑·' and '↓·' indeed (recursively) increase or decrease the quotedness levels of operator ides — but since bound variables are always used as is, they need to be always kept at a the 'zero' level. The evaluation fragments make sure that for every increment/decrement there is a corresponding decrement/increment that protects such bound instances.

- Because we know that the two operations cancel each other out, they can each be used as a protective wrapper for bound variable subterms in the evaluation of the other: the wrapper doesn't have to actually do its operation, it can simply serve as a future promise to cancel out when the process reaches that point.

- According to Theorem 5.2.29 (`unq` inverse of `q`), the evaluation fragment for '↑↓t' (6.3) needs to have a precondition that t is a representation of some term. See the description of '`up_down`' below for the way we deal with this.

- Note the restriction on the last rule (6.4): we cannot create a representation for an arbitrary value, only for canonical ones. (The corresponding '↓·' works only on shifted terms, which are all canonical.) In fact, we will not use it for

anything other than terms. The intuition here is that in the last two chapters we have been dealing with various facts about terms, but now we are working from within the system so we only deal with '`Term`'s. (It is possible to deal with other values, but again, this will reflect more than just syntax, falling outside the scope of this work.)

Additional `up`/`down`-related library objects

- '`down_up`' theorem — this is a simple theorem that states that $\downarrow\uparrow t = t$, which is proved using evaluation fragment 6.1.
- '`up_down`' theorem — as discussed above, evaluation fragment 6.3 will allow $\uparrow\downarrow t = t$ for all '`Term`'s. This theorem is later used instead of this fact, and it 'voluntarily' restricts this for '`Termof Term`'s, which is the type of '`Term`' representations.
- '`termin_member`' rule — this is the main rule[4] that makes the connection between '$\downarrow\in$ and '$\in$':

  $$\forall t : \mathsf{Term}, T : Univ_i.\ t \downarrow\in T \Rightarrow \downarrow(a) \in T$$

  Ideally, we'd want the other side to hold too, but as mentioned above, '$\in$' us used for well formedness in Nuprl, so there is no way to define it as a predicate. However, some restricted version of the other direction is needed — see Section 6.4 below.

A few rules and theorems and theorems establish various connections between '$\downarrow\in$', '`Termof`', '$\downarrow\cdot$', and '$\uparrow\cdot$':

- '`termof_member`' theorem — $\forall T : \mathbb{U}_i, t : \texttt{Termof}\ T.\ \downarrow t \in T$
- '`up_termin`' rule — $\forall t : \texttt{Term}.\ \uparrow t \downarrow\in \texttt{Term}$
- '`up_wf`' theorem — $\forall t : \texttt{Term}.\ \uparrow t \in \texttt{Term}$
- '`up_wf2`' theorem — $\forall t : \texttt{Term}.\ \uparrow t \in \texttt{Termof Term}$

Finally, '`term_downeq`' is defined as a relation that decides whether two terms represent the same value. Similarly to '$\downarrow\in$', it is displayed like Nuprl's '=', except that there is a '$\downarrow$' next to the '$\in$' symbol that denotes the type of the underlying equality: '$\cdot = \cdot \downarrow\in \cdot$'.

- '`term_downeq`' definition —
  $t_1 = t_2 \in T \ \equiv\ (t_1 \downarrow\in T\ \&\ t_2 \downarrow\in T)$ c& $\downarrow t1 = \downarrow t2 \in T$ (This definition uses Nuprl's 'conditional and' operator.)

[4] Actually, it is implemented as a theorem that is proved by '`Fiat`', but it should be converted to a rule. (It is easier to play around with theorems than it is to formalize rules.) A few more rules below are written this way.

- '`term_downeq_wf`' theorem — $\forall T : \mathbb{U}_i, t_1, t_2 : \texttt{Termof}\ T.\ (t_1 = t_2 \downarrow\in T) in \mathbb{P}_i$

The purpose of this relation is to demonstrate how easy it is to expose existing Nuprl functionality to the reflected user-accessible level.

6.3.7 `reps` part

This part defines a representation relation: '`reps`'. According to the same syntactic intuition[5], $x\,\texttt{reps}\,y \in T$ is displayed as $x \downarrow= y \in T$. The relation is defined as a simple abstraction using '$\downarrow\in$' and '$\downarrow\cdot$':

$$x \downarrow= y \in T \ \equiv\ x \downarrow\in T \text{ c\& } \downarrow x = y \in T$$

In addition, a '`repst`' is defined as '`reps`' with a hard-wired '`Term`' type, and displayed without the '$\in$ `Term`'. This is a convenience, given that most of our work is on '`Term`' denoting '`Term`'s.

A few well-formedness theorems are provided, and two theorems that establish the connection between this relation and '$\uparrow\cdot$':

- '`up_reps`' theorem — $\forall t : \texttt{Term}.\ \uparrow t \downarrow= t \in \texttt{Term}$
- '`up_repst`' theorem — $\forall t : \texttt{Term}.\ \uparrow t \downarrow= t$

Note that '`up_reps`' is also restricted to '`Term`'s, without this it is impossible to prove it since we will need to show that t is a term.

6.3.8 `term_subst` part

'`term_subst`' is a new term that represents substitutions, displayed as $\cdot[\cdot/\cdot]$ — remember that this is in Nuprl, so it is not the concrete substitution from the previous chapters. It is implemented by exposing Nuprl's existing '`fo_subst`' ML function, in a similar fashion to the technique that was used to implement '$\uparrow\cdot$' and '$\downarrow\cdot$': making '`fo_subst`' available as a user-accessible step-by-step version, which pushes substitutions into shifted operators.

There is an important issue that needs clarification before we describe the actual implementation of this new evaluation fragment. Substitution requires a (concrete) variable name to substitute — if we deal with *abstract syntax*, then we have no concrete names, and it seems like we must give up on having access to substitution! Indeed, Tarski's proof was constructed in a way that uses a concrete name, assuming this style of proofs will not be possible under an abstract syntax regimen.

For this we go back to the last paragraph of Section 4.3. In that paragraph it is explained that since the semantics of our abstract terms is based on α-equivalence

[5]Opinions vary about how intuitive this syntax is.

classes of concrete terms, then free variables are still accessible — and their semantics is of singleton sets of a single variable name, making a one-to-one correspondence between free variables names in the concrete and in the abstract worlds. Following this intuition, it is very much possible to use the Nuprl editor to quote a variable term, and the result, a shifted '`variable`' operator, is used as just a symbol. This is identical to the situation in Scheme: in an input syntax, symbols denote variables (bound identifiers), whereas *quoted symbols* are plain symbol values. Quoting a Scheme program turns all variable names into symbols, which is a result of its concrete syntax representation — the high-level hygienic macro system compensates for the inherent difficulties by using rewrite rules, and low-level macro facilities implement the high-level system by adding 'color' to symbols which makes it equivalent again to an abstract syntax. The proofs in the '`tarski`' theory rely heavily on this facility.

We now return to the problem of converting Nuprl's substitution to a step-by-step evaluation strategy. This problem is similar in its nature to the implementation of '$\uparrow\cdot$' and '$\downarrow\cdot$', but this case is more severe: we now get a 'black-box' function that performs substitution which we want to re-use (as described in Section 3.2.3, the actual substitution algorithm that Nuprl uses is complex due to user-interaction reasons). The main issue here is how do we deal with variable renaming. Our solution uses several (primitive) substitution steps to implement each step of our substitution. To avoid confusion, the following description does not use the '$\cdot[\cdot/\cdot]$' notation at all; instead, we use the existing ML function name (and argument syntax) for the built-in substitution, and '`term_subst(`t`;`v`;`e`)`' for our user level functionality (which is displayed as '$t[e/v]$').

The '`compute_term_subst`' function from '`reflection.ml`' does the following to evaluate '`term_subst(`t`;`v`;`e`)`':

1. Begin by evaluating t, the term we're substituting in, and v, the quoted variable (a 'symbol') that is to be replaced.

2. If either one is *not* a shifted operator, then term substitution is undefined, leave the term as is.

3. If $t = v$, return e.

4. If t is a quotation a different variable term, return it.

At this point, we know that t is a non-variable shifted term, and that v is a shifted variable. For example: $t =$ `op(x,y.`b`;z.`c`)` and $v =$ `x`, where the two subterms (b and c) might contain instances of both '`x`' and '`x`'. Our goal is to somehow use '`fo_subst`' to implement a single step of the

`term_subst(op(x,y.`b`;z.`c`); x;` e`)`

substitution. Obviously, we cannot simple unquote and use '`x`', since this will collide with the *real* '`x`' binder. The solution is to force Nuprl to perform the

necessary variable renamings — this will allow us to enjoy the fruits of a direct reflection: we will get Nuprl's sophisticated substitution algorithm for free, we can be quite sure that it is implemented correctly as it is at the core of the system, and users enjoy having the *same* consistent behavior on both the object and the meta levels.

We proceed with the following steps:

5. We force Nuprl to rename bindings by finding a new variable v that is not used in any of t's immediate bindings (using Nuprl's '`new_var`'), and constructing a new term t_1. Using our example:

$$t_1 = \underline{\texttt{op(x,y.}\langle b, v\rangle\texttt{;z.}\langle c, v\rangle\texttt{)}}$$

   (Note that the actual quoted name that is given to '`term_subst`' ('$\underline{\texttt{x}}$' in our example) is never used beyond the first steps above.)

6. Now use the built-in substitution with the new term and, the new variable, and the *same* term that is to be replaced:

$$t_2 = \texttt{fo\_subst}\ [v, e]\ t_1$$

7. We know that t_1 is not a variable term (it has the same structure as t), therefore t_2 must share the same top-level structure (same signature, which include its arities) of t and t_1, except that the substitution might force some renaming. Ignoring the actual subterms which are irrelevant, we get:

$$t_2 = \underline{\texttt{op(x}_1\texttt{,y}_1\texttt{.?;z}_1\texttt{.?)}}$$

8. In the above, the '$\texttt{x}_1$', '$\texttt{y}_1$', '$\texttt{z}_1$' variables will be the same as '`x`', '`y`', '`z`' except for some renaming that was forced to avoid capture. This is why we paired each subterm with the new variable: having the new variable forced Nuprl to rename identifiers that could be captured (if it wasn't there then Nuprl would choose to leave the names), and having the previous subterm maintains the same usage pattern of the bindings.

   It is now easy to see that if these new names are used instead of the original ones, *e.g.*:

$$\begin{aligned} \text{let } b_1 &= \texttt{fo\_subst}\ [\texttt{x}, \texttt{x}_1; \texttt{y}, \texttt{y}_1]\ b \\ \text{and } c_1 &= \texttt{fo\_subst}\ [\texttt{z}, \texttt{z}_1]\ c \\ \text{in } &\underline{\texttt{op(x}_1\texttt{,y}_1\texttt{.}b_1\texttt{;z}_1\texttt{.}c_1\texttt{)}} \end{aligned}$$

   then it is possible push the substitution one level inside since there are no name conflicts. The final result is therefore:

$$\underline{\texttt{op(x}_1\texttt{,y}_1\texttt{.term\_subst(}b_1\texttt{;}v\texttt{;}e\texttt{);z}_1\texttt{.term\_subst(}c_1\texttt{;}v\texttt{;}e\texttt{))}}$$

For example, say that we wish to push the '`term_subst`' in the following term (we now return to the easier way that these are displayed):

$$\forall x : \texttt{Term}.\ \ldots\ \underline{\texttt{foo(x,y.x+1)}}[\underline{\texttt{2+x}}/\underline{\texttt{x}}]\ \ldots$$

Using the above evaluation fragment, we get this:

$$\forall x : \texttt{Term}.\ \ldots\ \underline{\texttt{foo(x@0,y.x@0+1}}[\underline{\texttt{2+x}}/\underline{\texttt{x}}]\underline{\texttt{)}}\ \ldots$$

Note the naming scheme that resulted from Nuprl's substitution. Also note that it seems like no renaming is needed since there are no occurrences of the '$\underline{\texttt{x}}$' symbol in the body of the term — however, if this renaming would not have happened (*e.g.*, if we omit the v parts in step 5 above) then the rightmost x would be captured by the '$\underline{\texttt{foo}}$' instead of its original binding by the qualifier.

Additional `term_subst`-related library objects

- '`term_subst_wf`' rule — A 'rule' ('`Fiat`' theorem) that asserts '`term_subst`'s well-formedness. (Should be done with a standard '`term_substFormation`' and '`term_substEquality`'.)

- '`term_closed`' definition — this is defined as a simple abstraction in the following way:

  $$\texttt{term\_closed}(t)\ \equiv\ \forall s, v : \texttt{Term}.\ t[s/v] = t \in \texttt{Term}$$

  This is a nice benefit of exposing Nuprl's substitution mechanism: we exploit the fact that a system's substitution can be used to expose a host of related properties, and define some of these properties of Nuprl using our exposed substitution.

- '`free_in`' definition — an additional substitution-related property that is exposed through it (as a simple abstraction):

  $$\texttt{free\_in}(v; t)\ \equiv\ \neg\forall s : \texttt{Term}.\ t[s/v] = t \in \texttt{Term}$$

- '`term_closed_wf`' and '`free_in_wf`' theorems — demonstrating the well-formedness of the above two definitions.

- '`term_term_closed`' rule — this is another rule that is implemented as a '`Fiat`' theorem. It states that '`Termof Term`' is always closed. This reflects the fact that our (semantic) quotation function always produces a closed term since any free variables that it encounters are turned to symbols, as explained above. Note that when viewed from inside the system, there is no way to use any open terms (except for editor operations), which is why this fact (like many others that we have seen) talks about a second level of quotation.

- '`up_term_closed`' theorem — a quick lemma that states that

$$\forall t : \texttt{Term}.\ \texttt{term\_closed}(\uparrow t).$$

  It is proved using the previous rule, and needed by the '`tarski`' theory.

- '`free_iff_not_closed`' theorem — this theorem states that

$$\forall t : \texttt{Term}.\ (\exists v : \texttt{Term}.\ \texttt{free\_in}(v; t)) \Leftrightarrow \neg\texttt{term\_closed}(t)$$

  This theorem is quite involved, yet the '$\Leftarrow$' direction is incomplete: to be able to complete it, we would need a formalization of term induction.

Utilities

Finally, the '`reflection`' theory contains some utilities that demonstrate the how the color enhancements can be used for other interactions. Two examples that are provided implement a hypertext link that can pop up a given library object when clicked, and a 'hyper-term' that pops up its definition. These are used extensively by '`TarskiText`' (see below).

6.4 The '`tarski`' theory

The '`tarski`' theory demonstrates the interactive Nuprl version of the original concrete proof. The Nuprl version follows the concrete proof step by step and can be used to learn how the proof 'works' in any level of detail. It is especially instructive since it is a syntactic proof *about* syntax, therefore containing some very interesting pieces, including usages of '`term_subst`' and symbols (quoted free variables).

As a quick example, consider this little part of the original proof:

`q(t) reps t` [thus] `Q(t) reps q(r) if t reps r`

The claim here, translated to our syntax, is that since '$\uparrow t$' represents 't', then '$\updownarrow t$' represents '$\uparrow r$' if t represents r. This seems like it is correct (and it is), but here's a puzzle — is it true that the same holds by using two '$\uparrow\cdot$'s:

$$t\ \texttt{reps}\ r\ \Rightarrow\ \uparrow\uparrow\texttt{t reps}\uparrow\texttt{r}\ ?$$

Well, this turns out to be false, but it is not easy to realize unless one divorces oneself of all meaning and intuition and consider the formal rules only. But, now that we have the right Nuprl machinery, we can easily go and try proving this fact anyway, and see where we get stuck. The incomplete '`up_up_repst`' theorem demonstrates this — following this proof, we finally end up being required to show that $\uparrow\texttt{t} = \uparrow\texttt{r}$, when we know that $\downarrow\texttt{t} = \texttt{r}$, which is clearly false.

The Tarski proof that is presented in this theory requires two additional (similar) facts that are given as '`Fiat`' theorems: '`qup_termin`' and '`qsubx_termin`'.

These theorems demonstrate the need for some restricted version of the reverse of '`termin_member`', which would make these easy to show.

The main entry point to the '`tarski`' theory should be the '`TarskiText`' comment object, which is a hyper text version of the original proof. It is *extremely instructive* to follow the 'paper' proof alongside the interactive Nuprl proof — any reader of this text is strongly encouraged to try it out[6]. This will demonstrate the strong correspondence between the two proofs, as well allowing the reader to 'get a feeling' of how easy it is to use the system, and how complex syntactic games are reduced to easy games of symbol pushing.

[6]In case of technical difficulties, email '`eli@barzilay.org`'.

Chapter 7
Conclusions and Future Work

In this work we have demonstrated that syntax can be reflected efficiently and robustly using direct reflection. Using direct reflection as a design methodology has been demonstrated as a successful way to implement practical reflection in a simple and elegant manner, inherently robust, and cheap. This work has already been used as the foundation for MetaPRL's recent reflection mechanism [55] (still under active development).

7.1 Still Needed

As demonstrated by the proof in the '`tarski`' theory, the reflection system satisfies our expectations: it is easy to interact with, it is light-weight, and it is powerful enough to conveniently implement real syntactic-based proofs. However, there is still work that needs to be done complete it, as mentioned in various parts of Chapter 6.

- Additional facts need formal justification; Chapter 6 mentions several such items. Some of these are easy to achieve — *e.g.*, facts that require term induction were written before we had a formal account of term induction. Other facts require additional formal work, as they were not fully addressed in Chapter 4 and Chapter 5. An example of this is the theory's treatment of term substitution — '`term_subst`' of Section 6.3.8 — a proof of the described method is required.

  Such a proof raises another aspect of our work that applies to exposing other parts of the system to the user level: such a proof can go in two directions. First, we can add additional low-level facts about the system's behavior (proved externally), and describe higher-level functionality using those. The second approach, described in Chapter 3, is to expose high-level facts about system functionality, and simply assert that it is true — creating a form of a 'reflection by trust'.

  In general, the philosophical aspects of this work makes an argument for the second approach. This might seem like a bad choice, but consider term substitution: if the substitution algorithm in Nuprl is incorrect, then the system itself cannot be trusted for any formal use. The low-level justification of the substitution mechanism is therefore better placed if it is part of proving the system's correctness rather than justifying reflected functionality.

- Related to the last point, there are a few discrepancies between the material in the '`reflection`' theory and the formal account in Chapter 5. For example, the '`reps`' relation is not defined in the Nuprl theory same way that it is defined in Chapter 5. Additional work is needed to either modify the

theory definition, or provide a formal account of the validity of the existing definition.

- Some facts that were discussed in the formal parts of this work have not been included in the Nuprl library. Most noticeable in its absence is a term induction rule. Adding such a rule would not be too difficult, but it will have to be implemented in ML, as it also deals with terms of varying arities (remember that we define induction over substitution functions). The addition of an induction will also make it possible to prove additional material that is currently using Fiat.

7.2 Future Work

As for the future, the addition of reflection can be used for a wide variety of old and new problems, as well as enhancing the reflected capabilities.

- Syntactic reflection is only the first step towards a 'real' reflection rule: one that can be used to reflect the logic itself. Adding this will require exposing additional objects like sequents and tactics, and will allow meta proofs that could be used in various ways from simpler to more efficient proofs, as described in Knoblock's thesis [47], and by Allen, Constable, Howe and Aitken [6].

- While the current implementation seems sufficient, future proofs that will require significant syntactic functionality will be verbose to the point of being hard to manage. The current situation is similar to use Lisp's old '`defmacro`' which is less convenient for specifying complex syntax transformation rules. For such usages, it might be better to extend the system with a way of generating $\mathsf{Term} \rightarrow \mathsf{Term}$ transformation functions using a syntactic pattern-based rewrite mechanism. Again, such a mechanism would be easier to add to based on existing capabilities, so in the context of Nuprl, it might be better to enhance built-in functionality first, and in the context of systems like MetaPRL it will be natural to build on existing functionality.

- The syntactic capabilities of the system can be used for creating formal libraries that reason about *languages*. For example, theorem provers are commonly used to reason about properties of languages, and Nuprl is not an exception — including past and present projects that deal with syntax of various languages, from formalizing ML and Java, to custom languages that describe finite automatons when formalizing properties of network protocols. Such theories are much easier now that the main complexity that is involved in dealing with syntax and bindings is out of the way.

- Our experience demonstrates that we have the right tools that make exposing additional functionality easy. For example, note that '`term_downeq`' from

Chapter 6 is a simple addition that exposes reflected equality. It would be just as easy to expose additional system functionality, where each of these extensions opens a window to additional formal material that was either impossible or hard to discuss when working from within the system. For example, exposing the system's evaluation can make it feasible to build theories that discuss congruence, run-time complexity[1], and more.

- Our formal description and the functionality that is included in the '`reflection`' theory is currently restricted to terms that represent terms. If this is extended to any kind of terms, then Nuprl can be better used for reasoning about code-generation systems, including known problems such as an un-evaluation function that can construct a source syntax for an arbitrary function. This will involve justifying and using the '$\uparrow\cdot$' and '$\downarrow\cdot$' operators with fewer restrictions. '`up_reps`' demonstrates the flavor of such work — given a good extension, it might be possible to justify a rule that will work on a wider type range that we know how to lift.

- Finally, there is a lot of syntactic oriented meta-mathematical material that can be implemented in Nuprl, similar to the Tarski argument that is demonstrated by the '`tarski`' theory. Implementing such proofs in Nuprl serves as a great educational tool that can be used to learn about such proofs interactively, and it can be used for creating new formal content in an area that is notoriously confusing.

[1] Nuprl's evaluation mechanism works with a limit on the number of steps it can perform.

Appendix A
Glossary

The following is a glossary of some of the key terms used throughout this work. It should clarify usages of these terms that might be misinterpreted, due to the fact that their common usage is vague and/or ambivalent. In addition, due to the nature of discussing reflection, we make more distinction (*e.g.*, talking about different objects in different "levels") than common even in logic, formal languages, and computer science, therefore it is especially important to be precise. These should not be taken as absolute definitions, but as the intended meaning in the scope of this work — in an effort to help the reader.

The following is a relevant quote from Kleene [46]:

> When we are studying logic, the logic we are studying will pertain to one language, which we call the *object language*, because this language (including its logic) is an object of our study. Our study of this language and its logic, including our use of logic in carrying out the study, we regard as taking place in another language, which we call the *observer's language* (usually called "metalanguage" or "syntax language"). Or we may speak of the *object logic* and the *observer's logic*.
>
> It will be very important as we proceed to keep in mind this distinction between the logic we are studying (the object logic) and our use of logic in studying it (the observer's logic). To any student who is not ready to do so, we suggest that he close the book now, and pick some other subject instead, such as acrostics or beekeeping.

This work is an attempt to unify the meta and the object levels, while avoiding acrostics and beekeeping.

***Language*:** A notational system of communication, based on conventions. Used with some semantics, being computational, propositional, or iconic, or a combination of these.

***Formal Language*:** A language that has precise (algorithmic) formation and validation rules (syntactic definitions). Usually has some semantics as well.

***Term*:** The principal relata of the semantics of a language are terms. There are usually other names for this, depending on the nature of the language, for example: "words", "sentences", "expressions", and "formulas".

***Syntax*:** A method for analyzing terms structurally, usually designed to support some semantics.
Note: in some places "syntax" will be used instead of "syntactic object", this is done only when the context makes it clear (*e.g.*, "the numeral "1" is the syntax of the number one").

***Syntactic Object*:** Terms and other entities used by the syntactic analysis.

***Semantics*:** An interpretation that is used to explain core features of a practice of using possibly complex terms expressively.
Note: in some places, "semantics" is used as the result of an interpretation, in an way analogous for usage of function names (*e.g.*, "the square root of 2").

***Computational Semantics*:** A computational semantics specifies methods of computing with terms as programs or constituents of programs.

***Denotative Semantics*:** Semantics oriented towards truth interpretations.

***Semantic Object*:** An object in the domain of the semantic interpretation function.

***Object Level*:** The level of semantic objects a language or a system is dealing with. (For a computer system, this is the user level.)

***Meta Level*:** In the context of some system, the level of this system's description, syntactic or other. (For a computer system, this is the implementation level.)

***Object Language*:** The language provided for interaction with some system's object level.

***Meta Language*:** The language used for describing the meta-level.

***Representation*:** A relation, normally many-one, between entities for the purpose of reference or proxy. Reference would be using the object to "talk" about or "denote" the referent. Proxy data is used as a substitute for computation or reasoning that can then be used to make inferences about the represented entity. Frequently, this is used to refer to some representing entity, or the operation of finding it.

***Meaning/Reference*:** This is the inverse operation for representation, so if X represents Y, we say that Y is the meaning of X, or what X references. As with representation, the common case is in the context of syntax and semantics.

***Picture*:** Given a representation relation, a picture is a syntactic object that represents an entity in virtue of its having a structure similar to what it represents.

***Description*:** A description is a syntactic representation object of an entity in virtue of a complex (non-pictorial) analysis of its structure, *e.g.*, one that requires computation.

***Quotation*:** By quotation we mean representing a term by another term pictorially. This is usually either a proper quotation, or some form of a direct description when a proper quotation is impossible.

***Proper Quotation*:** This is a form of quotation where a term represents *itself*. This is always achieved by some context that changes the meaning of the term from its normal usage to representing itself.

***Direct Description*:** A syntactic description of a term that is non-complex in the sense that it is still a picture, but still cannot be considered a pictorial representation due to different structure than what it represents. This is sometimes used instead of quotations.

***Quasi-Quotation*:** A description of syntax that is made of a mix of quoted parts and non-direct descriptions (usually computational) of some parts. This can be taken as a quotation *template* where some of its parts are not quotations.

***Reflection*:** The phenomenon of unifying the meta-level with the object-level for some language (or system), allowing it to refer to itself. This occurs when the language becomes its *own* meta-language, and it depends on the features that are reflected. In this work we only deal with languages as the most important case of a reflected substrate system.
We distinguish two types of reflection:

- A *direct* reflection is one in which the representation of language features via its semantics is actually part of the definition of the semantics itself.
- A *indirect* reflection is one that is not direct. Typically this is achieved by a semantic description of the language within itself which requires an additional step demonstrating the correctness of this description.

***Reification*:** The process of converting an abstract meta-level entity to an object-level representation. In many places this is taken as the opposite operation to 'reflection', but in this work 'reflection' is used for the whole self-reference phenomenon, so this term is hardly used.

***Operator Shifting*:** In a syntax that is made of a recursive structure of operators, *shifting* an operator o_0 means using a different operator o_1 whose usage results in syntactic expression object denoting a usage of o_0. Usages of the shifted operator is similar to usages of the original operator, except their inputs are representations of inputs to the original.

***Concrete Syntax*:** Syntax representation that completely matches the actual input syntax, including any textual information that is later discarded. For this work, this is used only for syntax where binder names matter.

***Abstract Syntax*:** Syntax represented as data, similar to concrete syntax, except that irrelevant features are abstracted away. For this work, this is exactly like concrete syntax modulo alpha-equality. (Note that the "abstract" here is unrelated to functions, it is only to distinguished the represented syntax from the actual syntax.)

***Higher Order Abstract Syntax*, *HOAS*:** Similar to abstract syntax, except that binders are encoded using meta-level binders (usually functions).

Appendix B
Theory Files

The reflection theory is summarized in this appendix for reference. It is generated by Nuprl.

B.1 Reflection Theory

```
REFLECTION
----------

*C reflection_begin           ************ REFLECTION ************
*C begin_term                 ---------- term ----------
*D term                       EdAlias Term ::  Term== term
*R termFormation
              H ⊢ 𝕌 ext Term

                BY termFormation ()

                No Subgoals
*R termEquality
              H ⊢ Term = Term ∈ 𝕌

                BY termEquality ()

                No Subgoals
*M init_term   let TermEquality = (Refine ‘termEquality‘ []) ;;
              update_Trivial_additions ‘TermEquality‘ TermEquality ;;
*T term_wf                  1 Term ∈ 𝕌
*C begin_is_subst             ---------- is_subst ----------
*D is_subst0                  EdAlias is0 ::  is_subst0(.<t:term:E>)== is_subst(<t>)
*D is_subst1   EdAlias is1 ::  is_subst1(<x1:var>.<t:term:E>)== is_subst(<x1>.<t>)
*D is_subst2   EdAlias is2 ::
                is_subst2(<x1:var>,<x2:var>.<t:term:E>)
                == is_subst(<x1>,<x2>.<t>)
*D is_subst3   EdAlias is3 ::
                is_subst3(<x1:var>,<x2:var>,<x3:var>.<t:term:E>)
                == is_subst(<x1>,<x2>,<x3>.<t>)
*D is_subst4   EdAlias is4 ::
                is_subst4(<x1:var>,<x2:var>,<x3:var>,<x4:var>.<t:term:E>)
                == is_subst(<x1>,<x2>,<x3>,<x4>.<t>)
*D is_subst5   EdAlias is5 ::
                is_subst5(<x1:var>,<x2:var>,<x3:var>,<x4:var>,<x5:var>.<t:term:E>)
                == is_subst(<x1>,<x2>,<x3>,<x4>,<x5>.<t>)
*D is_subst6   EdAlias is6 ::
                is_subst6(<x1:var>,<x2:var>,<x3:var>,<x4:var>,<x5:var>,<x6:var>.<t:term:E>)
                == is_subst(<x1>,<x2>,<x3>,<x4>,<x5>,<x6>.<t>)
*D is_subst7   EdAlias is7 ::
                is_subst7(<x1:var>,<x2:var>,<x3:var>,<x4:var>,<x5:var>,<x6:var>,<x7:var>.
                <t:term:E>)
                == is_subst(<x1>,<x2>,<x3>,<x4>,<x5>,<x6>,<x7>.<t>)
*D is_subst8   EdAlias is8 ::
                is_subst8(<x1:var>,<x2:var>,<x3:var>,<x4:var>,<x5:var>,<x6:var>,<x7:var>,
                <x8:var>.<t:term:E>)
                == is_subst(<x1>,<x2>,<x3>,<x4>,<x5>,<x6>,<x7>,<x8>.<t>)
*R is_substFormation
              H ⊢ 𝕌 ext !call_ml{is_substFormation_extract_maker:s}

                BY is_substFormation !call_ml
```

```
                  Let SubGoals ext = !call_ml{is_substFormation_rule:s}
                SubGoals
*R is_substEquality
              H ⊢ t1 = t2 ∈ 𝕌 ext ext

                BY is_substEquality !call_ml

                  Let SubGoals ext = !call_ml{is_substEquality_rule:s}
                SubGoals
*C begin_term_eq              ---------- term_eq ----------
*D term_eq     EdAlias termeq ::
                if <a:Term_term>=<b:Term_term> then <s:term:*> else <t:term:E>
                == term_eq(<a>; <b>; <s>; <t>)
*R term_eqEquality
              H ⊢ if a1=b1 then s1 else t1 = if a2=b2 then s2 else t2 ∈ T

                BY term_eqEquality v

                H ⊢ a1 = a2 ∈ Term
                H ⊢ b1 = b2 ∈ Term
                H, v:(a1 = b1 ∈ Term) ⊢ s1 = s2 ∈ T
                H, v:((a1 = b1 ∈ Term) ⟶ Void) ⊢ t1 = t2 ∈ T
*R term_eqReduceTrue
              H ⊢ if a=b then s else t = u ∈ T

                BY term_eqReduceTrue ()

                H ⊢ s = u ∈ T
                H ⊢ a = b ∈ Term
*R term_eqReduceFalse
              H ⊢ if a=b then s else t = u ∈ T

                BY term_eqReduceFalse ()

                H ⊢ t = u ∈ T
                H ⊢ (a = b ∈ Term) ⟶ Void
*M init_term_eq
              let term_eqEquality p =
                (Refine ‘term_eqEquality‘ [mk_var_arg (new_invisible_var p)]
                 THENL
                 [Id; Id; Id;
                  OnLastHyp (\i. FoldAtAddr ‘false‘ [2] i
                                 THEN FoldTop ‘implies‘ i
                                 THEN FoldTop ‘not‘ i)]
                ) p
              ;;

              update_EqCD_additions ‘term_eqEquality‘ term_eqEquality;
*A eq_term                  x =t y ==  if x=y then tt else ff
*D eq_term_df               <a:term:*> =t <b:term:*>== eq_term(<a>; <b>)
*T eq_term_wf              0 ∀x,y:Term.  x =t y ∈ 𝔹
*M eq_term_eval
              let eq_term_evalC = UnfoldTopC ‘eq_term‘ ANDTHENC RedexC
              ;;
*T decidable__term_equal    2 ∀s,t:Term.  Dec(s = t ∈ Term)
*C begin_TermAuto             ---------- TermAuto ----------
*R isTerm      H ⊢ t = t ∈ Term

                BY isTerm ()

                H ⊢ is_subst0(.t)
*R isSubstTerm H ⊢ is_subst0(.t)
```

```
                BY isSubstTerm ()

                H ⊢ t ∈ Term
*R isSubst     H ⊢ t ext ext

                BY isSubst ()

                   Let SubGoals ext = !call_ml{is_subst_rule:s}
                SubGoals
*R isTermSubst H ⊢ t ext ext

                BY isTermSubst !call_ml

                   Let SubGoals ext = !call_ml{isTermSubst_rule:s}
                SubGoals
*R isSubstThin H ⊢ t ext ext

                BY isSubstThin !call_ml

                   Let SubGoals ext = !call_ml{is_substThin_rule:s}
                SubGoals
*C begin_termin_termof        ---------- termin/termof ----------
*A termin                     x ↓∈ t ==  !null_abstraction
*D termin_df   Parens ::Prec(atomrel)::
                {->0}<x:term>{<-}{\\? }↓∈ {->0}<t:type>{<-}
                == termin(<t>; <x>)
*R terminEquality
               H ⊢ (t1 ↓∈ A1) = (t2 ↓∈ A2) ∈ 𝕌

                BY terminEquality ()

                H ⊢ A1 = A2 ∈ 𝕌
                H ⊢ t1 = t2 ∈ Term
*R terminFormation
               H ⊢ 𝕌 ext t ↓∈ A

                BY terminFormation ()

                H ⊢ 𝕌 ext A
                H ⊢ Term ext t
*T termin_wf                2 ∀a:Term. ∀A:𝕌.  (a ↓∈ A) ∈ 𝕌
*M init_termin let terminEquality = (Refine ‘terminEquality‘ []) ;;
               update_Trivial_additions ‘terminEquality‘ terminEquality ;;
*A termof                     Termof A ==  {t:Term| t ↓∈ A}
*D termof_df                  Parens ::Prec(postop)::  Termof <t:Type:E>== termof(<t>)
*T termof_wf                1 ∀A:𝕌. Termof A ∈ 𝕌
*C begin_up_down              ---------- up/down ----------
*D up                         Parens ::Prec(preop)::  ↑ <t:term:E>== up(<t>)
*R upFormation H ⊢ Term ext ↑ t

                BY upFormation ()

                H ⊢ Term ext t
*R upEquality  H ⊢ ↑ t1 = ↑ t2 ∈ Term

                BY upEquality ()

                H ⊢ t1 = t2 ∈ Term
*D down                       Parens ::Prec(preop)::  ↓<t:term:E>== down(<t>)
*R downFormation
               H ⊢ A ext ↓t

                BY downFormation ()
```

```
                H ⊢ Termof A ext t
*R downEquality
              H ⊢ ↓t1 = ↓t2 ∈ A

                BY downEquality ()

                H ⊢ t1 = t2 ∈ Termof A
*T up_down                  2 ∀t:Termof Term. ↑ ↓t = t ∈ Term
*T down_up                  1 ∀t:Term. ↓↑ t = t ∈ Term
*T termin_member            0 ∀a:Term. ∀A:𝕌.  a ↓∈ A ⇒ ↓a ∈ A
*T termof_member            2 ∀A:𝕌. ∀a:Termof A.  ↓a ∈ A
*T up_termin                0 ∀x:Term. ↑ x ↓∈ Term
*T up_wf                    2 ∀x:Term. ↑ x ∈ Term
*T up_wf2                   2 ∀x:Term. ↑ x ∈ Termof Term
*A term_downeq                t1 = t2 ↓∈ A ==  (t1 ↓∈ A ∧ t2 ↓∈ A) c∧ (↓t1 = ↓t2 ∈ A)
*D term_downeq_df
              Parens ::Prec(atomrel)::Index 0 ::
                {[SOFT]<t1:term:L>{\\ }= {->0}<t2:term:L>{<-}{\\ }↓∈ {->0}<A:term:L>{<-}{]}
                == term_downeq(<A>; <t1>; <t2>)
*T term_downeq_wf           3 ∀A:𝕌. ∀x1,x2:Termof A.  (x1 = x2 ↓∈ A) ∈ ℙ
*C begin_reps                 ---------- reps ----------
*A reps                       x ↓= y ∈ t ==  (x ↓∈ t) c∧ (↓x = y ∈ t)
*D reps_df     Parens ::Prec(atomrel)::Index 0 ::
                {[SOFT]<x:term:L>{\\ }↓= {->0}<y:term:L>{<-}{\\ }∈ {->0}<t:term:L>{<-}{]}
                == reps(<t>; <x>; <y>)
*T reps_wf                  2 ∀u:𝕌. ∀t:Termof u. ∀x:u.  (t ↓= x ∈ u) ∈ ℙ
*A repst                      x ↓= y ==  x ↓= y ∈ Term
*D repst_df    EdAlias repsterm ::Parens ::Prec(atomrel)::Index 0 ::
                {[SOFT]<x:term:L>{\\ }↓= {->0}<y:term:L>{<-}{]}
                == repst(<x>; <y>)
*T reps_wf2                 2 ∀t,x:Term.  (t ↓= x ∈ Term) ∈ ℙ
*T repst_wf                 2 ∀t,x:Term.  (t ↓= x) ∈ ℙ
*T up_reps                  2 ∀t:Term. ↑ t ↓= t ∈ Term
*T up_repst                 2 ∀t:Term. ↑ t ↓= t
*C begin_term_subst           ---------- term_subst ----------
*D term_subst  EdAlias termsubst ::Parens ::Prec(atomrel)::Index 0 ::
                {[SOFT]<t:term:L>[{->0}<s:term:L>/<x:term:L>{<-}]
                == term_subst(<t>; <x>; <s>)
*T term_subst_wf            0 ∀t1,t2,t3:Term.  (t1[t2/t3]) ∈ Term
*A term_closed                term_closed(t) ==  ∀s,v:Term.  (t[s/v]) = t ∈ Term
*T term_closed_wf           2 ∀t:Term. term_closed(t) ∈ ℙ
*T term_term_closed         0 ∀t:Termof Term. term_closed(t)
*T up_term_closed           1 ∀t:Term. term_closed((↑ t))
*A free_in                    free_in(v; t) ==  ¬(∀s:Term. (t[s/v]) = t ∈ Term)
*T free_in_wf               2 ∀t,v:Term.  free_in(v; t) ∈ ℙ
#T free_iff_not_closed      3 ∀t:Term. (∃v:Term. free_in(v; t)) ⟺ ¬term_closed(t)
*C begin_utilities            ---------- utilities ----------
*D idform_color_df
              {BAD-COLOR(<type:type>,<c:color>)}== !dform_color{<type>:t, <c>:n}
              C<c:color-num>{== !dform_color{B:t, <c>:n}
              C+<c:color-num>{== !dform_color{B+:t, <c>:n}
              C-<c:color-num>{== !dform_color{B-:t, <c>:n}
              }<c:color-num>C== !dform_color{E:t, <c>:n}
              Q<c:quotedness>{== !dform_color{BQ:t, <c>:n}
              }<c:color-num>Q== !dform_color{EQ:t, <c>:n}
*D hypertext   EdAlias htext ::(HyperLink)::  C1O{<<t:text:*>>}1OC== !hyperlink{<t>:s}
*D hyperterm   EdAlias hterm ::(HyperLink)::  C1O{⌜<t:term:*>⌝}1OC== !hyperlink(<t>)
*C reflection_end             ************************************

============================== Theorems: Full Printout ==============================

<<< TERM_WF >>>
⊢ Term ∈ 𝕌
|
BY Unfold ‘member‘ 0
```

```
|
⊢ Term = Term ∈ 𝕌
|
BY TermEquality

<<< EQ_TERM_WF >>>
⊢ ∀x,y:Term.  x =t y ∈ 𝔹
|
BY Unfold ‘eq_term‘ 0 THEN Auto

<<< DECIDABLE__TERM_EQUAL >>>
⊢ ∀s,t:Term.  Dec(s = t ∈ Term)
|
BY Unfold ‘decidable‘ 0 THEN Auto
|
1. s: Term
2. t: Term
⊢ s = t ∈ Term ∨ ¬(s = t ∈ Term)
|
BY UseWitness ⌜if s=t then inl Ax  else (inr (λx.x) )⌝
THEN Auto
|
3. ¬(s = t ∈ Term)
⊢ (λx.x) ∈ ¬(s = t ∈ Term)
|
BY Unfold ‘not‘ 0 THEN Auto

<<< TERMIN_WF >>>
⊢ ∀a:Term. ∀A:𝕌.  (a ↓∈ A) ∈ 𝕌
|
BY GenUnivCD
|\
| 1. a: Term
| 2. A: 𝕌
| ⊢ (a ↓∈ A) ∈ 𝕌
| |
1 BY Unfold ‘member‘ 0
| |
| ⊢ (a ↓∈ A) = (a ↓∈ A) ∈ 𝕌
| |
1 BY Auto
|\
| 1. a: Term
| ⊢ 𝕌 ∈ 𝕌’
| |
1 BY Auto
 \
  ⊢ Term ∈ 𝕌
  |
  BY Auto

<<< TERMOF_WF >>>
⊢ ∀A:𝕌. Termof A ∈ 𝕌
|
BY GenUnivCD
|\
| 1. A: 𝕌
| ⊢ Termof A ∈ 𝕌
| |
1 BY Unfold ‘termof‘ 0 THEN Auto
 \
  ⊢ 𝕌 ∈ 𝕌’
  |
  BY Auto
```

```
<<< UP_DOWN >>>
⊢ ∀t:Termof Term. ↑ ↓t = t ∈ Term
|
BY GenUnivCD
|\
| 1. t: Termof Term
| ⊢ ↑ ↓t = t ∈ Term
| |
1 BY ComputeOpid ‘up‘ 0
| |
| ⊢ t = t ∈ Term
| |
1 BY D 1 THEN Auto
 \
  ⊢ Termof Term ∈ 𝕌
  |
  BY Auto

<<< DOWN_UP >>>
⊢ ∀t:Term. ↓↑ t = t ∈ Term
|
BY GenUnivCD
|\
| 1. t: Term
| ⊢ ↓↑ t = t ∈ Term
| |
1 BY ComputeOpid ‘down‘ 0 THEN Auto
 \
  ⊢ Term ∈ 𝕌
  |
  BY Auto

<<< TERMIN_MEMBER >>>
⊢ ∀a:Term. ∀A:𝕌.  a ↓∈ A ⇒ ↓a ∈ A
|
BY Fiat

<<< TERMOF_MEMBER >>>
⊢ ∀A:𝕌. ∀a:Termof A.  ↓a ∈ A
|
BY GenUnivCD THENA Auto
|
1. A: 𝕌
2. a: Termof A
⊢ ↓a ∈ A
|
BY D 2
|
2. a: Term
3. a ↓∈ A
|
BY BLemma ‘termin_member‘
THEN Auto

<<< UP_TERMIN >>>
⊢ ∀x:Term. ↑ x ↓∈ Term
|
BY Fiat

<<< UP_WF >>>
⊢ ∀x:Term. ↑ x ∈ Term
|
BY GenUnivCD
|\
| 1. x: Term
```

```
| ⊢ ↑ x ∈ Term
| |
1 BY Unfold ‘member‘ 0
THEN Refine ‘upEquality‘ []
THEN Auto
 \
  ⊢ Term ∈ 𝕌
  |
  BY Auto

<<< UP_WF2 >>>
⊢ ∀x:Term. ↑ x ∈ Termof Term
|
BY GenUnivCD
|\
| 1. x: Term
| ⊢ ↑ x ∈ Termof Term
| |
1 BY MemTypeCD THENA Auto
| |\
| | ⊢ ↑ x ∈ Term
| | |
1 2 BY Auto
|  \
|   ⊢ ↑ x ↓∈ Term
|   |
1   BY BLemma ‘up_termin‘ THENA Auto
 \
  ⊢ Term ∈ 𝕌
  |
  BY Auto

<<< TERM_DOWNEQ_WF >>>
⊢ ∀A:𝕌. ∀x1,x2:Termof A.  (x1 = x2 ↓∈ A) ∈ ℙ
|
BY GenUnivCD
|\
| 1. A: 𝕌
| 2. x1: Termof A
| 3. x2: Termof A
| ⊢ (x1 = x2 ↓∈ A) ∈ ℙ
| |
1 BY Unfold ‘term_downeq‘ 0
| |
| ⊢ (x1 ↓∈ A ∧ x2 ↓∈ A) c∧ (↓x1 = ↓x2 ∈ A) ∈ ℙ
| |
1 BY OnHyps [2;3] (Unfold ‘termof‘) THEN Auto
| |\
| | 2. x1: {t:Term| t ↓∈ A}
| | 3. x2: {t:Term| t ↓∈ A}
| | 4. x1 ↓∈ A
| | 5. x2 ↓∈ A
| | ⊢ ↓x1 ∈ A
| | |
1 2 BY BLemma ‘termin_member‘ THEN Auto
|  \
|   2. x1: {t:Term| t ↓∈ A}
|   3. x2: {t:Term| t ↓∈ A}
|   4. x1 ↓∈ A
|   5. x2 ↓∈ A
|   ⊢ ↓x2 ∈ A
|   |
1   BY BLemma ‘termin_member‘ THEN Auto
|\
| 1. A: 𝕌
| 2. x1: Termof A
```

```
| ⊢ Termof A ∈ 𝕌
| |
1 BY Auto
|\
| 1. A: 𝕌
| ⊢ Termof A ∈ 𝕌
| |
1 BY Auto
 \
  ⊢ 𝕌 ∈ 𝕌'
  |
  BY Auto

<<< REPS_WF >>>
⊢ ∀u:𝕌. ∀t:Termof u. ∀x:u.  (t ↓= x ∈ u) ∈ ℙ
|
BY GenUnivCD
|\
| 1. u: 𝕌
| 2. t: Termof u
| 3. x: u
| ⊢ (t ↓= x ∈ u) ∈ ℙ
| |
1 BY Unfold ‘reps‘ 0
| |
| ⊢ (t ↓∈ u) c∧ (↓t = x ∈ u) ∈ ℙ
| |
1 BY D 2 THEN Auto
| |
| 2. t: Term
| 3. t ↓∈ u
| 4. x: u
| 5. t ↓∈ u
| ⊢ ↓t ∈ u
| |
1 BY BLemma ‘termin_member‘ THEN Auto
|\
| 1. u: 𝕌
| 2. t: Termof u
| ⊢ u ∈ 𝕌
| |
1 BY Auto
|\
| 1. u: 𝕌
| ⊢ Termof u ∈ 𝕌
| |
1 BY Auto
 \
  ⊢ 𝕌 ∈ 𝕌'
  |
  BY Auto

<<< REPS_WF2 >>>
⊢ ∀t,x:Term.  (t ↓= x ∈ Term) ∈ ℙ
|
BY GenUnivCD
|\
| 1. t: Term
| 2. x: Term
| ⊢ (t ↓= x ∈ Term) ∈ ℙ
| |
1 BY Unfold ‘reps‘ 0
| |
| ⊢ (t ↓∈ Term) c∧ (↓t = x ∈ Term) ∈ ℙ
| |
1 BY Auto
```

```
| |
| 3. t ↓∈ Term
| ⊢ is_subst0(.(↓t))
| |
1 BY FLemma ‘termin_member‘ [3]
THEN Auto
|\
| 1. t: Term
| ⊢ Term ∈ 𝕌
| |
1 BY Auto
 \
  ⊢ Term ∈ 𝕌
  |
  BY Auto

<<< REPST_WF >>>
⊢ ∀t,x:Term.  (t ↓= x) ∈ ℙ
|
BY GenUnivCD
|\
| 1. t: Term
| 2. x: Term
| ⊢ (t ↓= x) ∈ ℙ
| |
1 BY Unfold ‘repst‘ 0
| |
| ⊢ (t ↓= x ∈ Term) ∈ ℙ
| |
1 BY BLemma ‘reps_wf2‘ THEN Auto
|\
| 1. t: Term
| ⊢ Term ∈ 𝕌
| |
1 BY Auto
 \
  ⊢ Term ∈ 𝕌
  |
  BY Auto

<<< UP_REPS >>>
⊢ ∀t:Term. ↑ t ↓= t ∈ Term
|
BY GenUnivCD THENA Auto
|
1. t: Term
⊢ ↑ t ↓= t ∈ Term
|
BY Unfold ‘reps‘ 0
|
⊢ (↑ t ↓∈ Term) c∧ (↓↑ t = t ∈ Term)
|
BY D 0
|\
| ⊢ ↑ t ↓∈ Term
| |
1 BY BLemma ‘up_termin‘ THENA Auto
 \
  2. ↑ t ↓∈ Term
  ⊢ ↓↑ t = t ∈ Term
  |
  BY BLemma ‘down_up‘ THENA Auto

<<< UP_REPST >>>
⊢ ∀t:Term. ↑ t ↓= t
|
```

```
BY GenUnivCD THENA Auto
|
1. t: Term
⊢ ↑ t ↓= t
|
BY Unfold ‘repst‘ 0
THEN BLemma ‘up_reps‘
THEN Auto

<<< TERM_SUBST_WF >>>
⊢ ∀t1,t2,t3:Term.  (t1[t2/t3]) ∈ Term
|
BY Fiat

<<< TERM_CLOSED_WF >>>
⊢ ∀t:Term. term_closed(t) ∈ ℙ
|
BY GenUnivCD
|\
| 1. t: Term
| ⊢ term_closed(t) ∈ ℙ
| |
1 BY Unfold ‘term_closed‘ 0
THEN Auto
 \
  ⊢ Term ∈ 𝕌
  |
  BY Auto

<<< TERM_TERM_CLOSED >>>
⊢ ∀t:Termof Term. term_closed(t)
|
BY Fiat

<<< UP_TERM_CLOSED >>>
⊢ ∀t:Term. term_closed((↑ t))
|
BY GenUnivCD THENA Auto
|
1. t: Term
⊢ term_closed((↑ t))
|
BY BLemma ‘term_term_closed‘
|
⊢ ↑ t ∈ Termof Term
|
BY Auto

<<< FREE_IN_WF >>>
⊢ ∀t,v:Term.  free_in(v; t) ∈ ℙ
|
BY GenUnivCD
|\
| 1. t: Term
| 2. v: Term
| ⊢ free_in(v; t) ∈ ℙ
| |
1 BY Unfold ‘free_in‘ 0 THEN Auto
|\
| 1. t: Term
| ⊢ Term ∈ 𝕌
| |
1 BY Auto
 \
  ⊢ Term ∈ 𝕌
  |
  BY Auto
```

```
<<< FREE_IFF_NOT_CLOSED >>> (partial)
⊢ ∀t:Term. (∃v:Term. free_in(v; t)) ⟺ ¬term_closed(t)
|
BY D 0 THENA Auto
|
1. t: Term
⊢ (∃v:Term. free_in(v; t)) ⟺ ¬term_closed(t)
|
BY D 0
|\
| ⊢ (∃v:Term. free_in(v; t)) ⇒ ¬term_closed(t)
| |
1 BY D 0 THENA Auto
| |
| 2. ∃v:Term. free_in(v; t)
| ⊢ ¬term_closed(t)
| |
1 BY D 0 THENA Auto
| |
| 3. term_closed(t)
| ⊢ False
| |
1 BY Unfold ‘term_closed‘ 3
| |
| 3. ∀s,v:Term.  (t[s/v]) = t ∈ Term
| |
1 BY D 2
| |
| 2. v: Term
| 3. free_in(v; t)
| 4. ∀s,v:Term.  (t[s/v]) = t ∈ Term
| |
1 BY Unfold ‘free_in‘ 3
| |
| 3. ¬(∀s:Term. (t[s/v]) = t ∈ Term)
| |
1 BY D 3
| |
| 3. ∀s,v:Term.  (t[s/v]) = t ∈ Term
| ⊢ ∀s:Term. (t[s/v]) = t ∈ Term
| |
1 BY D 0 THENA Auto
| |
| 4. s: Term
| ⊢ (t[s/v]) = t ∈ Term
| |
1 BY With ⌜s⌝ (D 3) THENA Auto
| |
| 3. s: Term
| 4. ∀v:Term. (t[s/v]) = t ∈ Term
| |
1 BY With ⌜v⌝ (D 4) THENA Auto
| |
| 4. (t[s/v]) = t ∈ Term
| |
1 BY Auto
 \
  ⊢ (∃v:Term. free_in(v; t)) ⇐ ¬term_closed(t)
  |
  BY D 0 THENA Auto
  |
  2. ¬term_closed(t)
  ⊢ ∃v:Term. free_in(v; t)
  |
  BY % To do this we need to know that the goal is decidable
  and this is only possible with induction. %
```

```
Id
  |
  |
  INCOMPLETE
```

B.2 Tarski Theory

```
TARSKI
------

*C tarski_begin              ************ TARSKI ************
*C TarskiText  We're assuming we have the type of terms (⌜Term⌝) and a representation
               relation between terms (⌜· ↓= ·⌝).

               a. We assume that if ⌜t ↓= s⌝ then ⌜t⌝ is closed.
                  <up_term_closed>

               Notation and some simple corrolaries (indicated by "thus"):
               (There are also assumptions about substitution into ⌜subx(·; ·)⌝ and ⌜↑ ·⌝)

               a1. ⌜x⌝ is a variable
               a2. ⌜subx(t; e)⌝ is substitution of term ⌜e⌝ for variable ⌜x⌝ in ⌜t⌝
               a3. ⌜t ↓= t' ∧ r ↓= r' ⇒ subx(t; r) ↓= subx(t'; r')⌝
                   <qsubx_repst>
               a4. ⌜subx(subx(t; r); e) = subx(subx(t; e); subx(r; e)) ∈ Term⌝
                   <qsubx_subx>
               a5. ⌜↑ t ↓= t⌝
                   <up_repst>
               a6. ⌜t ↓= r ⇒ ↑ t ↓= ↑ r⌝
                   <qup_repst>
                   (note that ⌜t ↓= r ⇒ ↑ ↑ t ↓= ↑ r⌝ does not work! <up_up_repst> -- Eli)

               b.  Thus, ⌜↑ ↑ t ↓= ↑ t⌝
                   <qup_up_repst>
               b1. ⌜subx((↑ t); e) = ↑ subx(t; e) ∈ Term⌝
                   <qup_subx>

               c1. ⌜f(t)⌝ is ⌜subx((↑ t); subx(x; (↑ x)))⌝
               c2. ⌜s(t)⌝ is ⌜subx(f(t); (↑ f(t)))⌝

               c3. Thus, ⌜s(t) = subx((↑ t); subx((↑ f(t)); (↑ ↑ f(t)))) ∈ Term⌝ by (a) on ⌜↑ t⌝
                   <s1>
               c4. Thus, ⌜s(t) ↓= subx(t; subx(f(t); (↑ f(t))))⌝ by (b)
                   <s2>
               c.  Thus, ⌜s(t) ↓= subx(t; s(t))⌝
                   <s_reps>

               d1. ⌜¬t⌝ is the term built from term ⌜t⌝ by the negation-denoting operator
               d.  Thus ⌜subx((¬t); e) = ¬subx(t; e) ∈ Term⌝
                   <qnot_subx>

               The Tarskian argument:

               Let ⌜RepsTruth(L; Tr; tr)⌝ where ⌜L⌝ and ⌜Tr⌝ are properties of terms
               and ⌜tr⌝ is a term, mean:
               1.  ⌜∀S:Term. (∃t:Term. S ↓= t) ⇒ L subx(tr; S)⌝
               2.  ⌜RespectsNot(Tr; L)⌝
                     meaning: ⌜∀t:Term. Tr ¬t ⟺ L t ∧ ¬(Tr t)⌝
               3.  ⌜ReflectsProp(Tr; tr; Tr)⌝
                     meaning: ⌜∀t,qt:Term.  qt ↓= t ⇒ {Tr subx(tr; qt) ⟺ Tr t}⌝
```

```
              This is meant to be part of the criterion for ⌜Tr⌝ being a truth predicate
              on ⌜L⌝, and for ⌜tr⌝ to denote ⌜Tr⌝ (in ⌜x⌝).

              <Tarski>
              Then there are no ⌜L⌝, ⌜Tr⌝, ⌜tr⌝ such that ⌜RepsTruth(L; Tr; tr)⌝ thus:
              4.  Assume ⌜RepsTruth(L; Tr; tr)⌝
                  {Tarski:t}
              5.  let ⌜S = s((¬tr)) ∈ Term⌝
                  {Tarski:t11}
              6.  ⌜S ↓= ¬subx(tr; S)⌝ by (5,c,d)
                  {Tarski:t111}
              7.  ⌜L subx(tr; S)⌝ by (4,1,6)
                  {Tarski:t1112}
              8.  ⌜Tr subx(tr; S) ⟺ Tr ¬subx(tr; S)⌝ by (4,3,6)
                  {Tarski:t11121}
              9.  ⌜Tr ¬subx(tr; S) ⟺ L subx(tr; S) ∧ ¬(Tr subx(tr; S))⌝ by (4,2)
                  {Tarski:t111211}
              10. ⌜Tr ¬subx(tr; S) ⟺ ¬(Tr subx(tr; S))⌝ by (9,7)
                  {Tarski:t1112111}
              *.  ⌜Tr subx(tr; S) ⟺ ¬(Tr subx(tr; S))⌝ by (8,10)
                  {Tarski:t11121111}
                  ... which is false so (4) is false.
*T qup_termin              0 ∀t:Term. t ↓∈ Term ⇒ ↑t ↓∈ Term
*T qsubx_termin            0 ∀t,r:Term.  t ↓∈ Term ∧ r ↓∈ Term ⇒ subx(t; r) ↓∈ Term
*T push_down_qup           2 ∀t:Term. t ↓∈ Term ⇒ ↓↑t = ↑ ↓t ∈ Term
*T push_down_qsubx         2
              ∀t,r:Term.
                subx(t; r) ↓∈ Term
                ⇒ t ↓∈ Term
                ⇒ r ↓∈ Term
                ⇒ ↓subx(t; r) = subx((↓t); (↓r)) ∈ Term
#T up_up_repst             3 ∀t,r:Term.  t ↓= r ⇒ ↑ ↑ t ↓= ↑ r
*T qup_repst               2 ∀t,r:Term.  t ↓= r ⇒ ↑t ↓= ↑ r
*T qup_up_repst            1 ∀t:Term. ↑↑ t ↓= ↑ t
*A subx                      subx(t; e) ==  t[e/x]
*T subx_wf                 2 ∀t,r:Term.  subx(t; r) ∈ Term
*T qsubx_repst             3 ∀t,r,t',r':Term.  t ↓= t' ∧ r ↓= r' ⇒ subx(t; r) ↓= subx(t'; r')
*T qsubx_subx              1
              ∀t,r,e:Term.  subx(subx(t; r); e) = subx(subx(t; e); subx(r; e)) ∈ Term
*T qup_subx                1 ∀t,e:Term.  subx((↑t); e) = ↑subx(t; e) ∈ Term
*T qnot_subx               1 ∀t,e:Term.  subx((¬t); e) = ¬subx(t; e) ∈ Term
*A f                         f(t) ==  subx((↑ t); subx(x; (↑ x)))
*T f_wf                    1 ∀t:Term. f(t) ∈ Term
*A s                         s(t) ==  subx(f(t); (↑ f(t)))
*T s_wf                    1 ∀t:Term. s(t) ∈ Term
*T s1                      3 ∀t:Term. s(t) = subx((↑ t); subx((↑ f(t)); (↑↑ f(t)))) ∈ Term
*T s2                      3 ∀t:Term. s(t) ↓= subx(t; subx(f(t); (↑ f(t))))
*T s_reps                  2 ∀t:Term. s(t) ↓= subx(t; s(t))
*A RespectsNot               RespectsNot(Tr; L) ==  ∀t:Term. Tr ¬t ⟺ L t ∧ ¬(Tr t)
*T RespectsNot_wf          2 ∀Tr,L:Term → ℙ.  RespectsNot(Tr; L) ∈ ℙ
*A ReflectsProp
              ReflectsProp(P; qP; Tr) ==  ∀t,qt:Term.  qt ↓= t ⇒ {Tr subx(qP; qt) ⟺ P t}
*T ReflectsProp_wf         2 ∀P:Term → ℙ. ∀qP:Term. ∀L:Term → ℙ.  ReflectsProp(P; qP; L) ∈ ℙ
*A RepsTruth   RepsTruth(L; Tr; tr) ==
                 (∀S:Term. (∃t:Term. S ↓= t) ⇒ L subx(tr; S))
                 ∧ RespectsNot(Tr; L)
                 ∧ ReflectsProp(Tr; tr; Tr)
*T RepsTruth_wf            2 ∀Tr:Term → ℙ. ∀tr:Term. ∀L:Term → ℙ.  RepsTruth(L; Tr; tr) ∈ ℙ
*T prop_and_iff            0 ∀A,B,C:ℙ.  B ⇒ (A ⟺ B ∧ C) ⇒ {A ⟺ C}
*T prop_iff_trans          0 ∀A,B,C:ℙ.  (A ⟺ B) ⇒ (B ⟺ C) ⇒ {A ⟺ C}
*T prop_iff_contra         0 ∀P:ℙ. (P ⟺ ¬P) ⇒ False
*T Tarski                  4 ¬(∃Tr:Term → ℙ. ∃tr:Term. ∃L:Term → ℙ. RepsTruth(L; Tr; tr))
*C tarski_end                ******************************

============================== Theorems: Full Printout ==============================
```

```
<<< QUP_TERMIN >>>
⊢ ∀t:Term. t ↓∈ Term ⇒ ↑_t ↓∈ Term
|
BY Fiat

<<< QSUBX_TERMIN >>>
⊢ ∀t,r:Term.  t ↓∈ Term ∧ r ↓∈ Term ⇒ subx(t; r) ↓∈ Term
|
BY Fiat

<<< PUSH_DOWN_QUP >>>
⊢ ∀t:Term. t ↓∈ Term ⇒ ↓↑_t = ↑ ↓t ∈ Term
|
BY GenUnivCD THENA Auto
|
1. t: Term
2. t ↓∈ Term
⊢ ↓↑_t = ↑ ↓t ∈ Term
|
BY ComputeOpid ‘down‘ 0
|
⊢ ↑ ↓t = ↑ ↓t ∈ Term
|
BY RWHL ‘up_down‘ 0
THEN Auto
THEN Unfold ‘termof‘ 0
THEN Auto

<<< PUSH_DOWN_QSUBX >>>
⊢ ∀t,r:Term.
  subx(t; r) ↓∈ Term ⇒ t ↓∈ Term ⇒ r ↓∈ Term ⇒ ↓subx(t; r) = subx((↓t); (↓r)) ∈ Term
|
BY GenUnivCD THENA Auto
|
1. t: Term
2. r: Term
3. subx(t; r) ↓∈ Term
4. t ↓∈ Term
5. r ↓∈ Term
⊢ ↓subx(t; r) = subx((↓t); (↓r)) ∈ Term
|
BY ComputeOpid ‘down‘ 0
|
⊢ subx((↓t); (↓r)) = subx((↓t); (↓r)) ∈ Term
|
BY Auto
|\
| ⊢ is_subst0(.(↓t))
| |
1 BY FLemma ‘termin_member‘ [4]
THEN Auto
 \
  ⊢ is_subst0(.(↓r))
  |
  BY FLemma ‘termin_member‘ [5]
THEN Auto

<<< UP_UP_REPST >>> (partial)
⊢ ∀t,r:Term.  t ↓= r ⇒ ↑ ↑ t ↓= ↑ r
|
BY GenUnivCD THENA Auto
|
1. t: Term
2. r: Term
3. t ↓= r
```

```
⊢ ↑ ↑ t ↓= ↑ r
|
BY OnHyps [3;0] (Unfold `repst`)
|
3. t ↓= r ∈ Term
⊢ ↑ ↑ t ↓= ↑ r ∈ Term
|
BY OnHyps [3;0] (Unfold `reps`)
|
3. (t ↓∈ Term) c∧ (↓t = r ∈ Term)
⊢ (↑ ↑ t ↓∈ Term) c∧ (↓↑ ↑ t = ↑ r ∈ Term)
|
BY D 3 THEN D 0
|\
| 3. t ↓∈ Term
| 4. ↓t = r ∈ Term
| ⊢ ↑ ↑ t ↓∈ Term
| |
1 BY BLemma `up_termin` THEN Auto
 \
  3. t ↓∈ Term
  4. ↓t = r ∈ Term
  5. ↑ ↑ t ↓∈ Term
  ⊢ ↓↑ ↑ t = ↑ r ∈ Term
  |
  BY RWHL `down_up` 0 THENA Auto
  |
  ⊢ ↑ t = ↑ r ∈ Term
  |
  BY Thin 5
  |
  |
  BY % Obviously bogus! % Id
  |
  |
  INCOMPLETE

<<< QUP_REPST >>>
⊢ ∀t,r:Term.  t ↓= r ⇒ ↑̲t ↓= ↑ r
|
BY GenUnivCD THENA Auto
|
1. t: Term
2. r: Term
3. t ↓= r
⊢ ↑̲t ↓= ↑ r
|
BY OnHyps [0;3] (Unfold `repst`)
|
3. t ↓= r ∈ Term
⊢ ↑̲t ↓= ↑ r ∈ Term
|
BY OnHyps [0;3] (Unfold `reps`)
|
3. (t ↓∈ Term) c∧ (↓t = r ∈ Term)
⊢ (↑̲t ↓∈ Term) c∧ (↓↑̲t = ↑ r ∈ Term)
|
BY D 3 THEN D 0
|\
| 3. t ↓∈ Term
| 4. ↓t = r ∈ Term
| ⊢ ↑̲t ↓∈ Term
| |
1 BY BLemma `qup_termin` THEN Auto
 \
  3. t ↓∈ Term
```

```
  4. ↓t = r ∈ Term
  5. ↑_t ↓∈ Term
  ⊢ ↓↑_t = ↑ r ∈ Term
  |
  BY RWHL ‘push_down_qup‘ 0 THENA Auto
  |
  ⊢ ↑ ↓t = ↑ r ∈ Term
  |
  BY HypSubst 4 0 THEN Auto

<<< QUP_UP_REPST >>>
⊢ ∀t:Term. ↑_↑ t ↓= ↑ t
|
BY GenUnivCD THENA Auto
|
1. t: Term
⊢ ↑_↑ t ↓= ↑ t
|
BY BLemma ‘qup_repst‘ THENA Auto
|
⊢ ↑ t ↓= t
|
BY BLemma ‘up_repst‘ THENA Auto

<<< SUBX_WF >>>
⊢ ∀t,r:Term.  subx(t; r) ∈ Term
|
BY GenUnivCD
|\
| 1. t: Term
| 2. r: Term
| ⊢ subx(t; r) ∈ Term
| |
1 BY Unfold ‘subx‘ 0 THEN Auto
|\
| 1. t: Term
| ⊢ Term ∈ 𝕌
| |
1 BY Auto
 \
  ⊢ Term ∈ 𝕌
  |
  BY Auto

<<< QSUBX_REPST >>>
⊢ ∀t,r,t',r':Term.  t ↓= t' ∧ r ↓= r' ⇒ subx(t;_r) ↓= subx(t'; r')
|
BY GenUnivCD THEN Auto
|
1. t: Term
2. r: Term
3. t': Term
4. r': Term
5. t ↓= t'
6. r ↓= r'
⊢ subx(t;_r) ↓= subx(t'; r')
|
BY OnHyps [0;5;6] (Unfold ‘repst‘)
|
5. t ↓= t' ∈ Term
6. r ↓= r' ∈ Term
⊢ subx(t;_r) ↓= subx(t'; r') ∈ Term
|
BY OnHyps [0;5;6] (Unfold ‘reps‘)
|
5. (t ↓∈ Term) c∧ (↓t = t' ∈ Term)
```

```
6. (r ↓∈ Term) c∧ (↓r = r' ∈ Term)
⊢ (subx(t; r) ↓∈ Term) c∧ (↓subx(t; r) = subx(t'; r') ∈ Term)
|
BY OnHyps [6;5;0] D
|\
| 5. t ↓∈ Term
| 6. ↓t = t' ∈ Term
| 7. r ↓∈ Term
| 8. ↓r = r' ∈ Term
| ⊢ subx(t; r) ↓∈ Term
| |
1 BY BLemma `qsubx_termin` THEN Auto
 \
  5. t ↓∈ Term
  6. ↓t = t' ∈ Term
  7. r ↓∈ Term
  8. ↓r = r' ∈ Term
  9. subx(t; r) ↓∈ Term
  ⊢ ↓subx(t; r) = subx(t'; r') ∈ Term
  |
  BY RWHL `push_down_qsubx` 0
THENA Auto
  |
  ⊢ subx((↓t); (↓r)) = subx(t'; r') ∈ Term
  |
  BY HypSubst 6 0 THEN HypSubst 8 0
THEN Auto

<<< QSUBX_SUBX >>>
⊢ ∀t,r,e:Term.  subx(subx(t; r); e) = subx(subx(t; e); subx(r; e)) ∈ Term
|
BY GenUnivCD THENA Auto
|
1. t: Term
2. r: Term
3. e: Term
⊢ subx(subx(t; r); e) = subx(subx(t; e); subx(r; e)) ∈ Term
|
BY Unfold `subx` 0
|
⊢ (subx(t; r)[e/x]) = subx((t[e/x]); (r[e/x])) ∈ Term
|
BY ComputeAtAddr [2] 0 THEN Auto

<<< QUP_SUBX >>>
⊢ ∀t,e:Term.  subx((↑ t); e) = ↑ subx(t; e) ∈ Term
|
BY GenUnivCD THENA Auto
|
1. t: Term
2. e: Term
⊢ subx((↑ t); e) = ↑ subx(t; e) ∈ Term
|
BY Unfold `subx` 0
|
⊢ (↑ t[e/x]) = ↑ (t[e/x]) ∈ Term
|
BY ComputeAtAddr [2] 0 THEN Auto

<<< QNOT_SUBX >>>
⊢ ∀t,e:Term.  subx((¬t); e) = ¬subx(t; e) ∈ Term
|
BY GenUnivCD THENA Auto
|
1. t: Term
2. e: Term
```

```
⊢ subx((¬t); e) = ¬subx(t; e) ∈ Term
|
BY Unfold ‘subx‘ 0
|
⊢ (¬t[e/x]) = ¬(t[e/x]) ∈ Term
|
BY ComputeAtAddr [2] 0 THEN Auto

<<< F_WF >>>
⊢ ∀t:Term. f(t) ∈ Term
|
BY GenUnivCD
|\
| 1. t: Term
| ⊢ f(t) ∈ Term
| |
1 BY Unfold ‘f‘ 0 THEN Auto
 \
  ⊢ Term ∈ 𝕌
  |
  BY Auto

<<< S_WF >>>
⊢ ∀t:Term. s(t) ∈ Term
|
BY GenUnivCD
|\
| 1. t: Term
| ⊢ s(t) ∈ Term
| |
1 BY Unfold ‘s‘ 0 THEN Auto
 \
  ⊢ Term ∈ 𝕌
  |
  BY Auto

<<< S1 >>>
⊢ ∀t:Term. s(t) = subx((↑ t); subx((↑ f(t)); (↑ ↑ f(t)))) ∈ Term
|
BY GenUnivCD THENA Auto
|
1. t: Term
⊢ s(t) = subx((↑ t); subx((↑ f(t)); (↑ ↑ f(t)))) ∈ Term
|
BY Assert ⌜s(t) = subx(f(t); (↑ f(t))) ∈ Term⌝
|\
| ⊢ s(t) = subx(f(t); (↑ f(t))) ∈ Term
| |
1 BY Unfold ‘s‘ 0 THEN Auto
 \
  2. s(t) = subx(f(t); (↑ f(t))) ∈ Term
  |
  BY Unfold ‘subx‘ 2
  |
  2. s(t) = (f(t)[↑ f(t)/x]) ∈ Term
  |
  BY ComputeWithTaggedTerm
  ⌜s(t) = ([0:f(t)][↑ f(t)/x]) ∈ Term⌝
  2
  |
  2. s(t) = (subx((↑ t); subx(x; (↑ x)))[↑ f(t)/x]) ∈ Term
  |
  BY Repeat (ComputeOpid ‘term_subst‘ 2)
  |
  2. s(t) = subx((↑ t[↑ f(t)/x]); subx((↑ f(t)); (↑ ↑ f(t)))) ∈ Term
  |
```

```
  BY Assert ⌜(↑ t[↑ f(t)/x]) = ↑ t ∈ Term⌝
  |\
  | ⊢ (↑ t[↑ f(t)/x]) = ↑ t ∈ Term
  | |
  1 BY BLemmaWithUnfolds ``term_closed`` `up_term_closed`
THEN Auto
   \
    3. (↑ t[↑ f(t)/x]) = ↑ t ∈ Term
    |
    BY HypSubst 3 2 THEN Auto

<<< S2 >>>
⊢ ∀t:Term. s(t) ↓= subx(t; subx(f(t); (↑ f(t))))
|
BY GenUnivCD THENA Auto
|
1. t: Term
⊢ s(t) ↓= subx(t; subx(f(t); (↑ f(t))))
|
BY RWHL `s1` 0 THENA Auto
|
⊢ subx((↑ t); subx((↑ f(t)); (↑_↑ f(t)))) ↓= subx(t; subx(f(t); (↑ f(t))))
|
BY BLemma `qsubx_repst` THEN Auto
|\
| ⊢ ↑ t ↓= t
| |
1 BY BLemma `up_repst` THEN Auto
 \
  ⊢ subx((↑ f(t)); (↑_↑ f(t))) ↓= subx(f(t); (↑ f(t)))
  |
  BY BLemma `qsubx_repst` THENA Auto
  |
  ⊢ ↑ f(t) ↓= f(t) ∧ ↑_↑ f(t) ↓= ↑ f(t)
  |
  BY D 0
  |\
  | ⊢ ↑ f(t) ↓= f(t)
  | |
  1 BY BLemma `up_repst` THEN Auto
   \
    ⊢ ↑_↑ f(t) ↓= ↑ f(t)
    |
    BY BLemma `qup_up_repst` THEN Auto

<<< S_REPS >>>
⊢ ∀t:Term. s(t) ↓= subx(t; s(t))
|
BY GenUnivCD THENA Auto
|
1. t: Term
⊢ s(t) ↓= subx(t; s(t))
|
BY ComputeWithTaggedTerm ⌜s(t) ↓= subx(t; [1:s(t)])⌝ 0
THENA Auto
|
⊢ s(t) ↓= subx(t; subx(f(t); (↑ f(t))))
|
BY BLemma `s2` THEN Auto

<<< RESPECTSNOT_WF >>>
⊢ ∀Tr,L:Term ⟶ ℙ.  RespectsNot(Tr; L) ∈ ℙ
|
BY GenUnivCD
|\
| 1. Tr: Term ⟶ ℙ
```

```
| 2. L: Term ⟶ ℙ
| ⊢ RespectsNot(Tr; L) ∈ ℙ
| |
1 BY Unfold ‘RespectsNot‘ 0 THEN Auto
|\
| 1. Tr: Term ⟶ ℙ
| ⊢ Term ⟶ ℙ ∈ 𝕌'
| |
1 BY Auto
 \
  ⊢ Term ⟶ ℙ ∈ 𝕌'
  |
  BY Auto

<<< REFLECTSPROP_WF >>>
⊢ ∀P:Term ⟶ ℙ. ∀qP:Term. ∀L:Term ⟶ ℙ.  ReflectsProp(P; qP; L) ∈ ℙ
|
BY GenUnivCD
|\
| 1. P: Term ⟶ ℙ
| 2. qP: Term
| 3. L: Term ⟶ ℙ
| ⊢ ReflectsProp(P; qP; L) ∈ ℙ
| |
1 BY Unfold ‘ReflectsProp‘ 0
THEN Unfold ‘guard‘ 0 THEN Auto
|\
| 1. P: Term ⟶ ℙ
| 2. qP: Term
| ⊢ Term ⟶ ℙ ∈ 𝕌'
| |
1 BY Auto
|\
| 1. P: Term ⟶ ℙ
| ⊢ Term ∈ 𝕌
| |
1 BY Auto
 \
  ⊢ Term ⟶ ℙ ∈ 𝕌'
  |
  BY Auto

<<< REPSTRUTH_WF >>>
⊢ ∀Tr:Term ⟶ ℙ. ∀tr:Term. ∀L:Term ⟶ ℙ.  RepsTruth(L; Tr; tr) ∈ ℙ
|
BY GenUnivCD
|\
| 1. Tr: Term ⟶ ℙ
| 2. tr: Term
| 3. L: Term ⟶ ℙ
| ⊢ RepsTruth(L; Tr; tr) ∈ ℙ
| |
1 BY Unfold ‘RepsTruth‘ 0 THEN Auto
|\
| 1. Tr: Term ⟶ ℙ
| 2. tr: Term
| ⊢ Term ⟶ ℙ ∈ 𝕌'
| |
1 BY Auto
|\
| 1. Tr: Term ⟶ ℙ
| ⊢ Term ∈ 𝕌
| |
1 BY Auto
 \
  ⊢ Term ⟶ ℙ ∈ 𝕌'
```

```
  |
  BY Auto

<<< PROP_AND_IFF >>>
⊢ ∀A,B,C:ℙ.  B ⇒ (A ⟺ B ∧ C) ⇒ {A ⟺ C}
|
BY Unfold ‘guard‘ 0 THEN ProvePropWith Auto

<<< PROP_IFF_TRANS >>>
⊢ ∀A,B,C:ℙ.  (A ⟺ B) ⇒ (B ⟺ C) ⇒ {A ⟺ C}
|
BY Unfold ‘guard‘ 0 THEN ProvePropWith Auto

<<< PROP_IFF_CONTRA >>>
⊢ ∀P:ℙ. (P ⟺ ¬P) ⇒ False
|
BY ProvePropWith Auto THEN Auto

<<< TARSKI >>>
⊢ ¬(∃Tr:Term ⟶ ℙ. ∃tr:Term. ∃L:Term ⟶ ℙ. RepsTruth(L; Tr; tr))
|
BY D 0 THENA Auto
|
1. ∃Tr:Term ⟶ ℙ. ∃tr:Term. ∃L:Term ⟶ ℙ. RepsTruth(L; Tr; tr)
⊢ False
|
BY Unfold ‘RepsTruth‘ 1 THEN ExRepD
|
1. Tr: Term ⟶ ℙ
2. tr: Term
3. L: Term ⟶ ℙ
4. ∀S:Term. (∃t:Term. S ↓= t) ⇒ L subx(tr; S)
5. RespectsNot(Tr; L)
6. ReflectsProp(Tr; tr; Tr)
|
BY Let ⌜S = s((¬tr)) ∈ Term⌝ THENA Auto
|
7. S: Term
8. S = s((¬tr)) ∈ Term
|
BY Assert ⌜S ↓= ¬subx(tr; S)⌝
|\
| ⊢ S ↓= ¬subx(tr; S)
| |
1 BY HypSubst (-1) 0 THENA Auto
| |
| ⊢ s((¬tr)) ↓= ¬subx(tr; s((¬tr)))
| |
1 BY RWHRevL ‘qnot_subx‘ 0 THEN Auto
| |
| ⊢ s((¬tr)) ↓= subx((¬tr); s((¬tr)))
| |
1 BY BLemma ‘s_reps‘ THEN Auto
 \
  9. S ↓= ¬subx(tr; S)
  |
  BY InstHyp [⌜S⌝] 4 THENW Auto
THENA (With ⌜¬subx(tr; S)⌝ (D 0) THEN Auto)
THEN Thin 4
  |
  4. RespectsNot(Tr; L)
  5. ReflectsProp(Tr; tr; Tr)
  6. S: Term
  7. S = s((¬tr)) ∈ Term
  8. S ↓= ¬subx(tr; S)
```

```
  9. L subx(tr; S)
  |
  BY Unfold ‘ReflectsProp‘ 5
THEN InstHyp [⌜¬subx(tr; S)⌝;⌜S⌝] 5
THENA Auto THEN Thin 5
  |
  5. S: Term
  6. S = s((¬tr)) ∈ Term
  7. S ↓= ¬subx(tr; S)
  8. L subx(tr; S)
  9. Tr subx(tr; S) ⟺ Tr ¬subx(tr; S)
  |
  BY Unfold ‘RespectsNot‘ 4
THEN InstHyp [⌜subx(tr; S)⌝] 4
THENA Auto THEN Thin 4
  |
  4. S: Term
  5. S = s((¬tr)) ∈ Term
  6. S ↓= ¬subx(tr; S)
  7. L subx(tr; S)
  8. Tr subx(tr; S) ⟺ Tr ¬subx(tr; S)
  9. Tr ¬subx(tr; S) ⟺ L subx(tr; S) ∧ ¬(Tr subx(tr; S))
  |
  BY Assert ⌜Tr ¬subx(tr; S) ⟺ ¬(Tr subx(tr; S))⌝
THENA (Using [‘B’,⌜L subx(tr; S)⌝]
             (BLemma ‘prop_and_iff‘)
       THEN Auto)
THEN OnHyps [9;7] Thin
  |
  7. Tr subx(tr; S) ⟺ Tr ¬subx(tr; S)
  8. Tr ¬subx(tr; S) ⟺ ¬(Tr subx(tr; S))
  |
  BY FLemma ‘prop_iff_trans‘ [7;8] THENA Auto
THEN OnHyps [8;7] Thin
  |
  7. Tr subx(tr; S) ⟺ ¬(Tr subx(tr; S))
  |
  BY FLemma ‘prop_iff_contra‘ [7] THEN Auto
```

BIBLIOGRAPHY

[1] Eric Aaron and Stuart Allen. Justifying calculational logic by a conventional metalinguistic semantics. Technical report, Cornell University, Ithaca, New York, September 1999. 3.3.4

[2] Klaus Aehlig. Normalization by evaluation does not need types. unpublished draft, January 2001. 6.3.6

[3] William Aitken. *Metaprogramming in Nuprl Using Reflection*. PhD thesis, Computer Science Dept., Cornell University, Ithaca, NY, 1994. Unfinished, not submitted. 2.3.5, 2.5.2, 3.2, 3.3.3, 2, 4.5

[4] Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987. 4.2

[5] Stuart F. Allen, Mark Bickford, Robert Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. Innovations in computational type theory using Nuprl. To appear in 2006. 2, 2.3

[6] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Symposium on Logic in Computer Science*, pages 95–197. IEEE, June 1990. 1, 2.3.5, 7.2

[7] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, August 2005. 1, 4.7

[8] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Symantics*, volume 103 of *Studies in Logic*. North-Holland, Amsterdam, 1981. 2.4

[9] Alan Bawden. Quasiquotation in lisp. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 4–12, 1999. 2

[10] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In B. Möller and J.V. Tucker, editors, *Prospects for Hardware Foundations*, volume 1546 of *LNCS*, pages 117–137. sv, 1998. 6.3.6

[11] R. S. Boyer and J. S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*, pages 103–84. Academic Press, New York, 1981. 20

[12] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics, second series*, 33:346–366, 1932. 2.3

[13] Alonzo Church. A set of postulates for the foundation of logic (second paper). *Annals of Mathematics, second series*, 34:839–864, 1933. 2.3

[14] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:55–68, 1940. 2.3

[15] Alonzo Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton University Press, Princeton, 1951. 2.3

[16] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986. 1, 2.2

[17] Robert L. Constable. Constructive mathematics as a programming logic I: Some principles of theory. In *Annals of Mathematics*, volume 24, pages 21–37. Elsevier Science Publishers, B.V. (North-Holland), 1985. 2.3

[18] Robert L. Constable. Naïve computational type theory. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability, Proceedings of International Summer School Marktoberdorf, July 24 to August 5, 2001*, volume 62 of *NATO Science Series III*, pages 213–260, Amsterdam, 2002. Kluwer Academic Publishers. 2, 2.3

[19] Robert L. Constable and Karl Crary. Computational complexity and induction for partial computable functions in type theory. In *Preprint*, 1998. 6.3.5

[20] Robert L. Constable and Karl Crary. Computational complexity and induction for partial computable functions in type theory. In Wilfried Sieg, Richard Sommer, and Carolyn Talcott, editors, *Reflections on the Foundations of Mathematics: Essays in Honor of Solomon Feferman*, Lecture Notes in Logic, pages 166–183. Association for Symbolic Logic, 2001. 6.3.5

[21] H. B. Curry, R. Feys, and W. Craig. *Combinatory Logic, Volume I*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1958. 2.3

[22] H. B. Curry, R. Feys, and W. Craig. *Combinatory Logic, Volume I*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1958. 2.4

[23] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Conf. Record 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'96*, pages 258–270. ACM Press, New York, 1996. 4.7.2

[24] N. G. de Bruijn. A survey of the project Automath. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980. 2.3

[25] Daniel de Rauglaudre. *CamlP4*, April 2000. 2.2, 2.4, 3.3, 3.3.6

[26] François-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *IJCAI'95 Workshop on Reflection and Metalevel Architectures and Their Applications in AI*, pages 29–38, August 1995. 2.2

[27] Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in coq. In M. Dezani and G. Plotkin, editors, *proceedings of the TLCA 95 Int. Conference on Typed Lambda Calculi and Applications*, volume 902, pages 124–138. Springer-Verlag LNCS, April 1995. 4.7.2, 4.7.2

[28] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In Philippe de Groote and J. Roger Hindley, editors, *proc. of the TLCA'97 Int. Conference*, pages 147–163. Springer-Verlag LNCS 1210, April 1997. An extended version is available as CMU Technical Report CMU-CS-96-172. 4.7.2

[29] M. Felleisen, Findler R. B., Flatt M., and Krishnamurthi S. The drscheme project: An overview. *SIGPLAN Notices: Functional Programming Column*, 1998. 2.3.3

[30] Matthias Felleisen. On the expressive power of programming languages. In N. Jones, editor, *ESOP '90 3rd European Symposium on Programming, Copenhagen, Denmark*, volume 432, pages 134–151. Springer-Verlag, New York, NY, 1990. 1

[31] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001. 4.7.2

[32] Kurt Gödel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover Publications, New York, 1992. 2

[33] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, NY, 1979. 2.2

[34] Andreas Paepcke Gregor Kiczales. Open implementations and metaobject protocols. Notes from a tutorial, 1995. 2.5.2

[35] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995. 1, 2.2

[36] J. Hickey, A. Nogin, R. L. Constable, B. E. Aydemir, E. Barzilay, Y. Bryukhov, R. Eaton, A. Granicz, A. Kopylov, C. Kreitz, V. N. Krupski, L. Lorigo, S. Schmitt, C. Witty, and X. Yu. MetaPRL — A modular logical environment. In David Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *LNCS*, pages 287–303. Springer-Verlag, 2003. 2.2, 2.5.2, 4.8, 5.2.23, 6.2

[37] Jason Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Cornell University, January 2001. 5.2.23

[38] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Symposium on Logic in Computer Science*, page 204. LICS, July 1999. 4.7, 4.7.2

[39] Furio Honsell and Marino Miculan. A natural deduction approach to dynamic logics. In *Proceedings of TYPES'95*, volume 1158 of *Lecture Notes in Computer Science*, pages 165–182, June 1995. workshop on proofs and types. 4.7.2

[40] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, NY, 1980. 2.3

[41] Douglas J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory*. PhD thesis, Cornell University, 1988. 2.3.5, c, 5.2.12

[42] Paul B. Jackson. *The Nuprl Proof Development System, Version 4.2 Reference Manual and User's Guide*. Cornell University, Ithaca, NY, January 1996. 3.2

[43] Guy L. Steele Jr. and Richard P. Gabriel. The evolution of Lisp. In Richard L. Wexelblat, editor, *History of Programming Languages: Proceedings of the ACM SIGPLAN Conference*, volume 28(3), pages 231–270. ACM Press, April 1993. 2.4

[44] Richard Kelsey, William D. Clinger, Jonathan Rees, et al. Revised^5 report on the algorithmic language scheme. *Journal of Higher Order and Symbolic Computation*, 11(1):7–105, 1998. 2.2, 2.3.1, 2.3.5, 3.1, 4.2

[45] G. Kiczales, desRivieres J., and Bobrow D. G. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991. 2.2, 2.5.2

[46] Stephen C. Kleene. *Mathematical Logic*. John Wiley & Sons, Inc., 1967. A

[47] T. Knoblock. *Mathematical Extensibility in Type Theory*. PhD thesis, Cornell University, 1987. 1, 2.3.5, 7.2

[48] H. Lauchli. An abstract notion of realizability for which intuitionistic predicate calculus is complete. In *Intuitionism and Proof Theory*, pages 227–34. North-Holland, Amsterdam, 1970. 2.3

[49] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland. 2.3

[50] J. McCarthy et al. *Lisp 1.5 Users Manual*. MIT Press, Cambridge, MA, 1962. 2.3.4

[51] John McCarthy. History of LISP. In Richard L. Wexelblat, editor, *History of Programming Languages: Proceedings of the ACM SIGPLAN Conference*, pages 173–197. Academic Press, June 1–3 1978. 2.3.4

[52] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997. 2.2

[53] Torben AE. Mogensen. Efficient self-interpretations in lambda calculus. *Journal of Functional Programming*, 2(3):345–363, 1992. 2.4

[54] Richard Montague. The proper treatment of quantification in ordinary english. In J. Hintikka, J. Moravcsik, and P. Suppes, editors, *Approaches to Natural Language*. Reidel, Dordrecht, 1973. 2.2

[55] Aleksey Nogin, Alexei Kopylov, Xin Yu, and Jason Hickey. A computational approach to reflective meta-reasoning about languages with bindings. In *Proceedings of MERLIN'05*, September 2005. Extended version was published as California Institute of Technology technical report CaltechCSTR:2005.003, available at `http://resolver.caltech.edu/CaltechCSTR:2005.003`. 2.5.2, 4.7.3, 4.8, 5.2.23, 7

[56] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford Sciences Publication, Oxford, 1990. 2.3

[57] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988. ACM Press. 4.1, 4.7, 4.7.3, 4.7.4

[58] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003. 4.7.2

[59] Andrew M. Pitts and Murdoch Gabbay. A metalanguage for programming with bound names modulo renaming. In *Mathematics of Program Construction*, pages 230–255, 2000. 4.1

[60] W. Van Orman Quine. *From a Logical Point of View: 9 Logico-Philosophical Essays*. Harvard University Press, Cambridge, Massachusetts, 1961. 2.4

[61] D. Scott. Constructive validity. In D. Lacombe M. Laudelt, editor, *Symposium on Automatic Demonstration*, volume 5(3) of *Lecture Notes in Mathematics*, pages 237–275. Springer-Verlag, New York, 1970. 2.3

[62] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 263–274. ACM Press, August 2003. 4.7.2

[63] C. Simonyi. Intentional programming: Innovation in the legacy age, 1996. 2.3.1

[64] B. C. Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, Massachusetts Institute of Technology, January 1982. 2.3.5

[65] B. C. Smith. Reflection and semantics in Lisp. *Principles of Programming Languages*, pages 23–35, 1984. 13, 2.3.5

[66] C. Smorynski. *Self-Reference and Modal Logic*. Springer-Verlag, Berlin, 1985. 2.5.2

[67] Raymond M. Smullyan. *Diagonalization and Self-Reference*. Number 27 in Oxford Logic Guides. Clarendon Press, Oxford, 1994. 2.2

[68] Paul Vincent Spade. Insolubles. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2005. 1

[69] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217, Amsterdam, The Netherlands, June 1997. ACM, New York. 2.4

[70] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, November 1999. 2.2, 2.4

[71] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. In M. Baaz, editor, *Computer Science Logic and 8th Kurt Gödel Colloquium (CSL'03 & KGC), Vienna, Austria. Proccedings*, Lecture Notes in Computer Science, pages 513–527. Springer-Verlag, Berlin, 2003. 4.7.3

[72] Christian Urban and Michael Norrish. A formal treatment of the barendregt variable convention in rule inductions. In *Proceedings of MERLIN'05*. ACM, September 2005. 4.7.2, 4.7.3

[73] Christian Urban and Christine Tasson. Nominal reasoning techniques in isabelle/hol. In *Proceedings of the 20th Conference on Automated Deduction (CADE 2005)*, volume 3632, pages 38–53, Tallinn, Estonia, July 2005. Springer Verlag. 4.7.2, 4.7.3

[74] Stephanie Weirich. *Programming With Types*. PhD thesis, Cornell University, 2002. 2.2