

**FROM SYNTACTIC THEORIES TO INTERPRETERS:  
SPECIFYING AND PROVING PROPERTIES**

by

**YONG XIAO**

**A DISSERTATION**

**Presented to the Department of Computer  
and Information Science  
and the Graduate School of the University of Oregon  
in partial fulfillment of the requirements  
for the degree of  
Doctor of Philosophy**

**June 2004**

“From Syntactic Theories to Interpreters:

Specifying and Proving Properties,” a dissertation prepared by Yong Xiao in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science. This dissertation has been approved and accepted by:



---

Dr. Zena M. Ariola, Chair of the Examining Committee

May 27, 2004  
Date

Committee in charge:

Dr. Zena M. Ariola, Chair  
Dr. Andrzej Proskurowski  
Dr. Christopher Wilson  
Dr. Sergey Yuzvinsky

Accepted by:



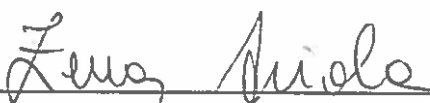
---

Dean of the Graduate School

©2004 Yong Xiao

An Abstract of the Dissertation of  
Yong Xiao for the degree of Doctor of Philosophy  
in the Department of Computer and Information Science  
to be taken June 2004

Title: FROM SYNTACTIC THEORIES TO INTERPRETERS:  
SPECIFYING AND PROVING PROPERTIES

Approved:   
Dr. Zena M. Ariola

Syntactic theories have been developed to reason about many aspects of modern programming languages[AB97, AFM<sup>+</sup>95, LS97, SS99, FLS99]. Having roots in the  $\lambda$ -calculus, these theories rely on transforming source programs to other source programs. Only the *syntax* of the programming language is relevant.

Experience shows that the development of such theories is error-prone. In order to ensure that the theories are sensible, many properties need to be checked. For example, we need to know if there is always a rule to rewrite a valid program and if a unique value can be obtained (which is described by a unique-decomposition lemma) and if the type of a program is preserved during evaluation (which is described by a subject reduction lemma). In many situations, the proofs of these properties do not require deep insight. However, many purported proofs suffer from being incomplete and often the missed case is the most problematic one. Thus, we think that in order to rely on syntactic theories, it is of mandatory importance to design tools that support their development. The work described in this thesis offers a first step in that direction.

We introduce the specification language SL which can directly reflect primitive notions of syntactic theories such as evaluation contexts and dynamic constraints. An experimental system has been implemented that generates interpreters from SL specifications. Currently the generated interpreters are programs in CAML[Cam], a dialect in the ML family. Various examples have been tested, including the operational semantics of core-ML and a syntactic theory for Verilog[FLS99].

We experiment with the SL system to automatically check whether the unique-decomposition and subject reduction properties hold for the specified syntactic theories.

We map the unique-decomposition lemma to the problems of checking equivalence and ambiguity of syntactic definitions. Because checking these properties of context-free grammars is undecidable, we work with regular tree grammars and use algorithms on finite tree automata to perform the checking. To make up for the insufficient expressiveness of regular tree grammars, we enhance the basic framework with built-in types, constants, contexts, and polymorphic types.

In order to specify type systems, we extend the SL system to allow rules written in natural semantics. We also introduce a meta-level of the SL system with utilities to handle substitution and equivalence, and to simulate pattern matching and type reasoning. A subject reduction lemma can be represented as a meta-theorem. The proof is performed automatically by induction on the rewriting rules, the type checking rules, as well as the structure of the object-terms. The induction step consists of instantiating the meta-theorem into simpler forms which are also meta-theorems. The base case corresponds to the meta-theorems that can be easily proved, for example, a logical tautology. The induction framework for automatically proving subject reduction can be easily extended to other properties.

## CURRICULUM VITA

NAME OF AUTHOR: Yong Xiao

### GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon  
University of Science and Technology of China

### DEGREES AWARDED:

Doctor of Philosophy in Computer and Information Science,  
2004, University of Oregon  
Master of Science in Computer Science and Technology,  
1996, University of Science and Technology of China  
Bachelor of Science in Computer Science and Technology,  
1993, University of Science and Technology of China

### AREAS OF SPECIAL INTEREST:

Programming Languages  
Software Engineering

### PROFESSIONAL EXPERIENCE:

Research Assistant, Department of Computer Science and Technology,  
University of Science and Technology of China, Hefei, China, 1993–1996  
Research Assistant, Computer and Information Science,  
University of Oregon, Eugene, Oregon, 1996–2001  
Senior R&D Engineer,  
Synopsys, Sunnyvale, California, 2001–2004

## ACKNOWLEDGEMENTS

I own too much thanks to my advisor, Professor Zena M. Ariola, who set me on this track and kept me moving forward. Zena had given valuable guidance and offered financial support for my studying and research. She helped to set a coloboration between Cristal group at INRIA, France and the programming language team at University of Oregon. The project was to generate interpreters from language specification, which the initial part of the dissertation (Chapter II). Zena mentored the follow up projects which include automatically proving properties and enhancement on the initial system. The efforts result in Chapter III, Chapter IV, and Chapter V in the dissertation. This work is result of Zena's care, patience, encouragement, and important suggestions.

Thanks to my committee, Professor Andrzej Proskurovsky, Professor Christopher Wilson, and Professor Sergey Yuzvinsky. Andrzej and Chris have also been in my committee for DRP and Area exam. I appreciate their comments and suggestions on the drafts.

I would like to thank Michel Mauny, Xavier Leroy and Didier Rémy of the Cristal Group in INRIA for their contribution to the earlier version of the SL system. I am also indebted to Amr Sabry for his joint work on Chapter III and Miley Semmelroth for his detailed review on my dissertation, as well as other faculty, staff, and students of the Computer and Inforamtion Science Department who provided help, advice, and discussion.

Thanks to my parents, my wife, and my daughter, who had always encouraged me and supported me.

## TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION . . . . .	1
Syntactic Theories . . . . .	2
Logical Frameworks . . . . .	7
The SL System . . . . .	15
Organization . . . . .	16
II. THE SL SYSTEM – GENERATING INTERPRETERS FROM SYNTACTIC THEORIES . . . . .	18
Overview of the SL System . . . . .	18
The SL Language . . . . .	22
Compiling SL specifications . . . . .	26
The SL Runtime Libraries . . . . .	44
Example of code generated by SL . . . . .	51
Summary and Discussion . . . . .	58
III. AUTOMATICALLY PROVING DECOMPOSITION . . . . .	61
Manual Proof of Unique Decomposition . . . . .	63
Equivalence and Ambiguity of Grammars . . . . .	64
Regular Tree Grammars and Finite Tree Automata . . . . .	65
Algorithms for Proving Unique Decomposition . . . . .	72
Extensions . . . . .	79
Automatically proving unique-decomposition in the SL System . . . . .	90
Summary and Discussion . . . . .	93
IV. EXTENSIONS TO THE SL SYSTEM . . . . .	94
Abstract patterns . . . . .	94
Natural Deduction Rules . . . . .	102
Signature Functions . . . . .	106
Meta-substitution . . . . .	117
Summary and Discussion . . . . .	120



	Page
V. AUTOMATICALLY PROVING SUBJECT REDUCTION . . . . .	122
A Manual Proof of Subject Reduction . . . . .	123
Representing the Type System in SL . . . . .	124
A Meta-layer for the SL System . . . . .	125
An Example of Automatically Proving Subject Reduction by Induction	134
Automatically Proving Subject Reduction by Induction . . . . .	138
Summary and Discussion . . . . .	141
VI. CONCLUSION AND FUTURE WORK . . . . .	143
Contributions and Applications of the SL System . . . . .	143
Future work . . . . .	145
APPENDIX	
THE SL LANGUAGE . . . . .	147
BIBLIOGRAPHY . . . . .	152

## LIST OF FIGURES

Figure	Page
1. Typing rules for $\lambda^{\rightarrow}$ . . . . .	6
2. Deriving the type of a $\Lambda$ -term . . . . .	6
3. Deriving the type of a $\Lambda$ -term . . . . .	7
4. An SL Specification of a Simple CBV Language . . . . .	19
5. Abstract syntax of the SL language . . . . .	23
6. Generating CAML code . . . . .	45
7. Transformation between runtime terms and lambda terms . . . . .	47
8. An automaton for the grammar of Example 2 . . . . .	73
9. SL Specification of Decomposing a Simple CBV Language . . . . .	91
10. An SL Specification of a Simple CBV Language using abstract patterns . . . . .	96
11. call-by-value lambda calculus in natural deduction semantics . . . . .	102
12. An SL Specification of a Simple CBV Language specified with natural deduction rules . . . . .	104
13. An SL Specification of a Simple CBV Language using signature functions	108
14. An SL Specification of a Simple CBV Language using signature functions	113
15. An SL Specification of a Simple CBC Language with meta-substitution . . . . .	119
16. The SL specification of the type system of CBV $\lambda$ -calculus . . . . .	126
17. Meta-layer of the SL system . . . . .	127
18. Unification of meta-expressionsxb . . . . .	130
19. Merging two environments . . . . .	131
20. Meta-pattern matching . . . . .	133

21. Meta-expression evaluation . . . . . 135

## LIST OF TABLES

Table	Page
1. The type checking rules for type expressions . . . . .	27
2. The type checking rules for SL expressions . . . . .	28
3. The type checking rules for SL patterns . . . . .	30
4. The type checking rules for declarations . . . . .	31
5. The type checking rules for axioms . . . . .	32
6. The type checking rules for inference rules . . . . .	33
7. The type checking rules for abstract patterns . . . . .	99
8. The type checking rules for absPAT definitions . . . . .	100
9. The type checking rules for extended inference rules . . . . .	105
10. The type checking rules for sigfun definitions . . . . .	110
11. The type checking rules for sigfun axioms . . . . .	114
12. The type checking rules for sigfun inference rules . . . . .	115

## CHAPTER I

### INTRODUCTION

In everyday computing we deal with a variety of programming languages. Some languages are for general purposes such as C, C++, Ada, ML, Prolog, Lisp, and Java. Some are designed for specific domains, such as SQL for querying, Perl for text processing, Tex for documenting, and Verilog for hardware description. Moreover, new languages emerge everyday.

There are numerous aspects to programming language design and implementation. First are issues of concrete syntax, abstract syntax, and parsing. The next question concerns the semantics of a program, such as its type and its result by evaluation. In addition, many questions regarding properties of the language can be asked. Is it effectively decidable if an input expression is well-typed? Are types preserved during the execution of a program? Does every well-typed program produce a unique result?

For each programming language, implementing a compiler or interpreter and proving its properties is a daunting task. It is highly desirable to develop a framework that isolates the uniformities of a wide class of language designs so that much of the effort can be expended once and for all. By convention, we call the framework the *meta-system*, and call the languages *object-languages*.

In this thesis, we present a special-purpose framework that is tailored to specifying and proving properties of programming languages. Our framework is called the SL system. It provides necessary primitive notions, such as data types, dynamic values,

contexts, axioms, and inference rules, for specifying the syntax and semantics of languages. It also deals with properties of a language, such as decomposition and subject reduction. The SL system automatically generates compilers from the specifications, and it automatically proves the property statements or shows counterexamples about the statements.

The rest of this chapter is divided as follows: We first give an introduction to syntactic theories, one of the common formal representations for languages. We then provide a survey of logic frameworks. Next, we give a brief overview of the SL system. Finally, we outline the organization of the rest of this thesis.

### Syntactic Theories

The notion of *syntactic theories* is one of the most well-known and extensively used formal description techniques for programming languages [AB97, AFM<sup>+</sup>95, LS97, SS99, FLS99]. Having roots in the  $\lambda$ -calculus [Plo75], these theories rely on transforming source programs to other source programs. Only the *syntax* of the programming language is relevant. In this section, we use the call-by-value  $\lambda$ -calculus as a running example to introduce operational semantics based on syntactic theories.

### Operational Semantics

The set of terms of the call-by-value  $\lambda$ -calculus is generated inductively over an infinite set of variables ranged over by  $x$ ,  $y$ , etc, and it includes  $\lambda$ -abstractions and applications, defined as follows:

$\Lambda$ -Terms  $M ::= x \mid \lambda x.M \mid MM$

Two terms that only differ in the names of bound variables are considered identical, e.g.,  $\lambda x.x$  is equivalent to  $\lambda y.y$ . This is called  $\alpha$ -conversion.

The operational semantics is based on the  $\beta_v$  reduction rule which requires a syntactic definition of the notion of value:

Values  $V ::= \lambda x.M$   
 $\beta_v \quad (\lambda x.M)V \rightarrow [V/x]M$

where  $[V/x]M$  is the term resulting from substituting  $V$  for the free occurrences of the variable  $x$  in  $M$ .

A call-by-value computation consists of successively applying the  $\beta_v$  reduction rule to a subterm. Positions of  $\beta_v$  redexes are restricted by an evaluation context which is defined as follows:

$E ::= \square \mid EM \mid VE$

where  $\square$  represents a “hole”. If  $E$  is an evaluation context, then  $E[M]$  denotes the term that results from placing  $M$  in the hole of  $E$ . The evaluation of a program is then defined by a stepping relation, denoted by  $\mapsto$ , given as follows:

$$\frac{M \rightarrow M'}{E[M] \mapsto E[M']}$$

In other words, at each step of evaluation, we attempt to *decompose* a program

into an evaluation context and a *redex* (a term matching the left-hand side of  $\beta_v$  rule) and to rewrite the redex using our axiom.

For example, the term  $(\lambda y.y) ((\lambda x.x) \lambda z.z)$  can be reduced to  $(\lambda y.y) \lambda z.z$  by using the  $\beta_v$  rule and the evaluation context  $(\lambda y.y)\square$ . It can be further reduced to  $\lambda z.z$  by using the  $\beta_v$  rule and the evaluation context  $\square$ . The term  $\lambda z.z$  is a *normal form*, that is, a term that cannot be further reduced. Note that in the first reduction, the term  $(\lambda x.x) \lambda z.z$  has to be reduced to value before the  $\beta_v$  can be applied to the outmost term.

### Unique Decomposition

The evaluation process defined above is not guaranteed to yield a final value. For instance, the term  $y(\lambda x.x)$  cannot be decomposed into an evaluation context filled with a redex. The existence of programs that have no decomposition indicates either a mistake in the definitions, or situations that need to be accounted for using compile-time or run-time checks. In both cases, it is useful to have a precise statement that lists all the possible decompositions for a term: every term can be decomposed as a value (*i.e.*, a final answer that needs no further evaluation) or an evaluation context filled with a current instruction. The current instruction could be a valid redex, or could indicate an erroneous program that contains a reference to an unbound variable. The formal definitions are given below:

(Values)  $V ::= \lambda x.M$

(Redexes)  $R ::= (\lambda x.M)V$

(Decomposed Terms)  $D ::= V \mid E[R] \mid E[x]$



The existence of decompositions for all terms guarantees that evaluation considers all possible cases for the input programs. But for many languages, it is important that the evaluation is specified deterministically. In other words, the decomposition of each program is unique. This property is stated as follows:

Lemma 1 (Unique Decomposition)

Every term  $M$  can be uniquely decomposed into one of the following forms: a value  $V$ , or a demand for a variable  $x$  within an evaluation context  $E$ , or a redex  $R$  within an evaluation context  $E$ .

### Subject Reduction

Most programming languages are equipped, either explicitly or implicitly, with type systems. We define a simple typed extension (denoted by  $\lambda^\rightarrow$ ) of the call-by-value  $\lambda$ -calculus previously presented as follows:

$$\begin{array}{ll} \Lambda\text{-terms} & M ::= x \mid \lambda x : T. M \mid M_1 M_2 \\ \text{Types} & T ::= a \mid T \rightarrow T \end{array}$$

where  $\alpha$  denotes a type variable and  $T_1 \rightarrow T_2$  stands for a function type whose argument type is  $T_1$  and its result type is  $T_2$ .  $\rightarrow$  is right-associate, that is,  $T_1 \rightarrow T_2 \rightarrow T_3$  is the same as  $T_1 \rightarrow (T_2 \rightarrow T_3)$ .

Some  $\Lambda$ -terms are meaningless. For example, it is nonsensical to apply a term which is not a function. We will consider *well-typed*  $\Lambda$ -terms, those that can be derived from the type-checking rules. The type-checking rules for the simply-typed  $\lambda$ -calculus are given in Figure 1, where  $\Gamma$  stands for a set of type assignments for constants and

$$\begin{array}{l}
\text{(start)} \quad \frac{}{\Gamma \vdash x : \tau} \quad \text{if } (x : \sigma) \in \Gamma \\
\text{(\(\rightarrow\))I} \quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \\
\text{(\(\rightarrow\))E} \quad \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}
\end{array}$$

FIGURE 1. Typing rules for  $\lambda^{\rightarrow}$ 

$$\frac{\frac{\frac{\frac{}{\{x : \alpha, y : \beta\} \vdash x : \alpha} \text{start}}{\{x : \alpha\} \vdash \lambda y. x : \beta \rightarrow \alpha} \rightarrow\text{I}}{\{\} \vdash \lambda x. \lambda y. x : \alpha \rightarrow \beta \rightarrow \alpha} \rightarrow\text{I}}{\{\} \vdash (\lambda x. \lambda y. x) \lambda z. z : \beta \rightarrow \tau \rightarrow \tau} \rightarrow\text{I}}{\{\} \vdash (\lambda x. \lambda y. x) \lambda z. z : \beta \rightarrow \tau \rightarrow \tau} \rightarrow\text{I}}$$

FIGURE 2. Deriving the type of a  $\Lambda$ -term

variables.  $\Gamma$  has the following properties:

- *weakening*: Assignments can be added freely.
- *strengthening*: Unused assignments can be removed.
- *permutation*: The order of assignments is irrelevant.
- *contraction*: The same assignment can be used more than once.

As an example, the term  $(\lambda x. \lambda y. x) \lambda z. z$  can be assigned the type  $\beta \rightarrow \tau \rightarrow \tau$ .

Figure 2 shows how this type is derived using the type system in Figure 1.

An important property of the type system is *subject reduction*, that is, the evaluation of a well-typed program preserves its type.

$$\frac{\frac{\{y : \beta, z : \tau\} \vdash z : \tau}{\{y : \beta\} \vdash \lambda z. z : \tau \rightarrow \tau} \rightarrow I}{\{\} \vdash \lambda y. \lambda z. z : \beta \rightarrow \tau \rightarrow \tau} \rightarrow I$$

FIGURE 3. Deriving the type of a  $\Lambda$ -term

### Lemma 2 (Subject Reduction)

If  $M \rightarrow M'$  and  $\Gamma \vdash M : \sigma$ , then  $\Gamma \vdash M' : \sigma$ .

For example, the term  $(\lambda x. \lambda y. x) \lambda z. z$  can be reduced to  $\lambda y. \lambda z. z$  which also has type  $\beta \rightarrow \tau \rightarrow \tau$ . The type derivation of the term  $\lambda y. \lambda z. z$  is given in Figure 3.

In this thesis, we introduce a special logical framework that assists in writing syntactic theories and proving some important properties such as unique decomposition and subject reduction. Next is a brief survey of logical frameworks.

### Logical Frameworks

Logical frameworks are systems for specifications of deductive systems. The semantics and properties of a programming language can be described using various deductive systems that are given via axioms and rules of inference. For example, a type system is specified by type-checking or type-inference rules and the result of a program is determined by evaluation rules. Properties are also proved in a certain reasoning system. Therefore, we can interchange the following concepts: *programming languages*, *logics*, and *deductive systems*.

In this section, we give a brief overview of the general themes, concepts, and design choices for logical frameworks.

## Representing Syntax

The first task of a logical framework is to represent the syntax of object-languages. In order to concentrate on the structure of expressions, we do not consider issues of concrete syntax and parsing, and only focus on specifying abstract syntax.

### First-order Representation

One of the simplest approach of representing syntax is to choose the first-order terms themselves. For our running example, the representation of a variable  $x$  is its name with a tag, written as  $\text{Var}(x)$ . An abstraction,  $\lambda x.t$ , is represented as an application of a first-order constructor  $\text{Lam}$  to a pair of  $x$  and the representation of  $t$ . Similarly, the applications in the object-language are represented as terms with constructor  $\text{App}$ . We use  $\llbracket - \rrbracket$  for the representation function that maps object expressions to meta-expressions.

$$\begin{aligned}\llbracket x \rrbracket &= \text{Var}(x) \\ \llbracket \lambda x.M \rrbracket &= \text{Lam}(x, \llbracket M \rrbracket) \\ \llbracket M_1 M_2 \rrbracket &= \text{App}(\llbracket M_1 \rrbracket, \llbracket M_2 \rrbracket)\end{aligned}$$

Systems such as Coq [DFH<sup>+</sup>93] allow first-order inductive definitions. This approach is similar to the representation above.

```
datatype o = Var of string | Lam of string * o | App of o * o
```

It has the advantage of easily handling sub-typing. First-order inductively defined types are regular tree types. Membership of a term in such a type can be decided by a finite

tree automaton [GS84] and sub-typing can be reduced to tree automata homomorphism.

The first-order representation above does not support the variable convention. Renaming of bound variables must be implemented explicitly. For example,  $\text{Lam}(x, \text{Var}(x))$  and  $\text{Lam}(y, \text{Var}(y))$  need to be identified. Another issue of the first-order representation is that substitution must be also explicitly provided in the specification.

### Higher-Order Abstract Syntax

The approach of higher-order abstract syntax (HOAS) represents variables of an object-language by variables in the meta-language. Most HOAS approaches, such as those for LF [HHP93] and Isabelle [Isa, Pau90], are supported by the simply-typed  $\lambda$ -calculus. Notice that this  $\lambda$ -calculus works as a meta-calculus.

We use a type constant  $\circ$  for the object  $\lambda$ -terms. We also introduce constants assigned to certain types, and represent object-terms in  $\Lambda$ -terms as follows:

$$\begin{aligned} \llbracket x \rrbracket &= x \\ \llbracket \lambda x M \rrbracket &= \text{Lam}(\lambda x : \circ. \llbracket M \rrbracket) & \text{Lam} &: (\circ \rightarrow \circ) \rightarrow \circ \\ \llbracket M_1 M_2 \rrbracket &= \text{App} \llbracket M_1 \rrbracket \llbracket M_2 \rrbracket & \text{App} &: \circ \rightarrow \circ \rightarrow \circ \end{aligned}$$

The first and second cases depict HOAS representation. The variables are used as both object-variables and meta-variables, and object lambda abstractions are represented as applications of a constant to a meta-abstraction. The bound object variables are bound meta-variables. Therefore, object  $\alpha$ -conversion automatically holds in this representation. Moreover, substitution in the object-language is handled by  $\beta$ -conversion in the meta-language. For example,  $(\lambda x : \circ. \llbracket M_1 \rrbracket) \llbracket M_2 \rrbracket = \llbracket \llbracket M_2 \rrbracket / x \rrbracket \llbracket M_1 \rrbracket = \llbracket \llbracket M_2 / x \rrbracket M_1 \rrbracket$ .

The latter equation is proved by simple induction on  $\Lambda$ -terms.

An object-term can be represented by many  $\Lambda$ -terms that can be converted to each other. In order to make the representation non-ambiguous, only *canonical forms* are chosen as representations. Canonical forms have the form of  $\lambda x_1 : \tau_1 \dots \lambda x_n : \tau_n. M_0 M_1 \dots M_n$  where  $M_0$  is either a constant or a variable,  $M_0 M_1 \dots M_n$  is fully applied, i.e.,  $M_0 M_1 \dots M_n$  should not be of a function type, and  $M_1, \dots, M_n$  are canonical  $\Lambda$ -terms. The canonical form of a  $\Lambda$ -term is unique; it is obtained by first repeatedly applying  $\beta$ -conversion from left to right and then repeatedly applying  $\eta$ -conversion from left to right. The canonical forms are also called  *$\beta\eta$ -long normal forms*.

### Representing Semantics

The next step is to represent the semantics of the object-language. Semantic rules can be represented as rewriting rules, logical formulae, or dependent types.

#### Rewriting Rules

If the object-terms are represented as first-order meta-terms, it is only natural to represent semantic rules as first-order rewriting rules, also called *small-step semantic rules*. The Elan system adopts this approach. For example, the evaluation rules for the call-by-value  $\lambda$ -calculus are represented as follows:

$$\begin{aligned} \text{App}(\text{Lam}(x, M_1), v) &\rightarrow [v/x]M_1 \quad \text{where } v = \text{Lam}(y, M_3) \\ E[M] &\rightarrow E[M'] \quad \text{where } M \rightarrow M' \end{aligned}$$

### Natural Deduction Rules

The operational semantics can be also formalized in *natural deduction rules*, also called *big-step semantics rules*.  $\mapsto$  defines the result of evaluation (in multiple steps).

$$\begin{array}{l}
 \text{(evlam)} \quad \frac{}{\lambda x.M \mapsto \lambda x.M} \\
 \\
 \text{(evapp)} \quad \frac{M_1 \mapsto \lambda x.M'_1 \quad M_2 \mapsto V_2}{M_1 M_2 \mapsto [V_2/x]M'_1}
 \end{array}$$

### Logic Formulas

The semantic rules can be also represented as higher-order formula, or *Hereditary Harrop Formulas*[Mil89]. Hereditary Harrop Formulas are defined as follows:

$$\text{Hereditary Harrop Formulas } H ::= P \mid \top \mid H_1 \wedge H_2 \mid H_1 \supset H_2 \mid \forall x : T.H$$

where  $T$  stands for a type. The Isabelle [Isa, Pau90] system adopts this approach. Hereditary Harrop formula has two important differences with Horn logic formula: the addition of types so that quantifiers now range over simply-typed  $\lambda$ -terms, and the fact that generalization allows the body of clauses to contain implication and universal quantifications (so called *embedded implication* and *embedded universal quantification*).

A statement about the evaluation of a term (such as “ $M$  evaluates to  $V$ ”) is represented by a meta-expression (such as  $\text{eval}(M, V)$ ), where  $\text{eval}$  is a predicate in the meta-logic. Each inference rule defining a case of  $\text{eval}$  is turned into an axiom in the meta-language. For our running example, the natural semantics can be represented as

follows:

$$\text{(evlam)} \quad \forall f : \mathbf{o} \rightarrow \mathbf{o}. \text{eval}(\text{Lam } f, \text{Lam } f)$$

$$\text{(evapp)} \quad \forall m_1 : \mathbf{o}. \forall m_2 : \mathbf{o}. \forall v : \mathbf{o}.$$

$$(\forall f : \mathbf{o} \rightarrow \mathbf{o}. \forall v_2 : \mathbf{o}.$$

$$\text{eval}(m_1, \text{Lam } f) \wedge \text{eval}(m_2, v_2) \wedge \text{eval}(f \ v_2, v) \Rightarrow$$

$$\text{eval}(\text{App } m_1 \ m_2, v)$$

where “ $\forall$ ”, “ $\wedge$ ”, and “ $\Rightarrow$ ” denote quantification, conjunction, and implication, respectively.

### Dependent Types

Another approach is to design a representation of derivations themselves explicitly in the meta-language. More specifically, we introduce a new type for evaluations in the object-language, where each rule becomes a constructor of the new type. This approach, which is related to the Curry-Howard isomorphism [How80], is known as judgments-as-types and derivations-as-objects. System LF [Gar92] represents evaluation relations as types in the dependently typed  $\lambda$ -calculus,  $\lambda^\Pi$ , and represents reductions of terms as objects.

$\lambda^\Pi$  is a calculus with three levels (kinds, types, and objects) which are defined as follows:

$$\text{Kinds} \quad K ::= \text{TYPE} \mid \Pi x : T. K$$

$$\text{Types} \quad T ::= \alpha \mid a \mid T \ M \mid \Pi x : T. T$$

$$\text{Objects} \quad M ::= c \mid x \mid \lambda x : T. M \mid M \ M$$



where `TYPE` is a kind constant standing for all types in  $\lambda^+$ .

For our running example, the evaluation can be represented as a dependent type, `eval`, indexed by object programs before and after evaluation. `eval m v` is a type indicating that `m` evaluates to `v`. The two inference rules are represented as constructors for building terms in the evaluation type. Building terms using the constructors then captures evaluation using the inference rules. Details are below.

```

eval    :  $\Pi m : o. \Pi v : o. \text{TYPE}$ 
evalam  :  $\Pi f : o \rightarrow o. \text{eval}(\text{Lam } f)(\text{Lam } f)$ 
evapp   :  $\Pi m_1 : o. \Pi m_2 : o. \Pi v : o.$ 
          $(\Pi f : o \rightarrow o. \Pi v_2 : o.$ 
            $\text{eval } m_1 (\text{Lam } f) \rightarrow \text{eval } m_2 v_2 \rightarrow \text{eval } (f v_2) v) \rightarrow$ 
            $\text{eval } (\text{App } m_1 m_2) v$ 

```

### Representing and Proving Properties

Since logical frameworks are designed to express the language and inference rules of deductive systems at a very high level of abstraction, one rightly suspects that they should be amenable to an investigation of the meta-theory of deductive systems, such as the decomposition property.

One approach is to represent properties as propositions in some mathematical system such as set theory or automata theory, and to apply theorems in the system to prove the properties. The logical framework must embed the mathematical theorems.

A more common approach is to represent properties as logical formulae so that proving the properties corresponds to checking validity of the logic formulae. This tech-

nique, used in systems such as Nuprl [C<sup>+</sup>86, Nup] and  $\lambda$ Prolog [IPr, NM88], involves proof searching.

Another common approach is using inductive proof, both over the structure of expressions and derivations. Thus, one naturally looks towards frameworks that permit inductive definitions of judgments and support the corresponding induction principles. Unfortunately, induction conflicts with the representation technique of higher-order abstract syntax. For example, the lambda abstraction was represented in LF by the constructor

$$\text{Lam} : (\text{o} \rightarrow \text{o}) \rightarrow \text{o}$$

Because of the negative occurrence of  $\text{o}$  in the type of  $\text{Lam}$ , attempts to formulate a valid induction principle for the type  $\text{o}$  would fail.

Several options have been explored to escape this dilemma. The first, for example used in [BBKM93, MN94] is to reject the notion of higher-order abstract syntax and use inductive representation directly. This engenders a complication of the encoding and consequently of the meta-theory, which now has to deal with many lemmas regarding variable naming. Using de Bruijn indices alleviates this problem somewhat. In Coq, users can define functions over inductively defined types. Another possibility is to externalize the induction. This leads to a three-level architecture: the object-logic, the logical framework, and a meta-framework for reasoning about the logical framework. Schurmann and Pfenning's Twelf system [Twe] is one such framework.

Twelf incorporates HOAS and induction by analyzing different cases of canonical normal forms of meta-terms. An inductive function can be encoded as a logical relation. More specifically, a function  $f$  in the type of  $T_1 \rightarrow T_2$  may be represented as a relation  $f : T_1 \rightarrow T_2 \rightarrow \text{TYPE}$ . As an example, consider the meta-theorem that whenever

$m \mapsto v$  is derivable, then  $v$  is a value. The theorem can be represented as a type:

$$vs : \prod m : o. \prod v : o. \text{eval } m \ v \rightarrow \text{value } v \rightarrow \text{TYPE}$$

where `value` is a type  $\prod v : o. \text{TYPE}$  which has only one constant `lamvalue` :  $\prod f : o \rightarrow o. \text{value } (\text{Lam } f)$ .

In summary, logical frameworks are meta-systems for specifying deductive systems. Most existing logical frameworks are designed to be general purpose. When used to specify syntactic theories, they do not support the necessary primitive semantic notions and they lack support for the automation of proofs. To address these issues, we developed a new system called SL.

### The SL System

The SL system employs a first-order representation of object-languages. It uses first-order conditional rewriting rules (*i.e.*, axioms and inferences) for interpreters, and natural deduction rules for type systems. The SL system supports values and contexts as built-in notions leading to great flexibility and simplicity in specifying semantic rules. Various examples have been tested including the operational semantics of core-ML (lambda-calculus with built-in operations, store operations, and exception handling), type inference for core-ML, a syntactic theory for a store encapsulation language[SS99], and a syntactic theory for Verilog[FLS99].

We extend the SL system to automatically check whether unique decomposition and subjective reduction hold in the specified syntactic theories.

We map the unique-decomposition lemma to the problems of checking equiv-

alence and ambiguity of syntactic definitions. Because checking these properties of context-free grammars is undecidable, we use regular tree grammars and algorithms on finite tree automata to perform the checking. To make up for the insufficient expressiveness of regular tree grammars, we extend the basic framework with built-in types and constants, contexts, and polymorphic types.

In order to specify type systems, we enhance the SL system to allow rules written in natural semantics. We also introduce a meta-level of the SL system with utilities to handle substitution and equivalence, and to simulate pattern matching and type reasoning. The subject reduction lemma is represented as a meta-level theorem, with a list of meta-expressions as premises and a list of meta-expressions as conclusions. The proof is done automatically by induction on the rewriting rules, the type checking rules, as well as the structure of the data types defining the object-language. The induction step consists of instantiating the meta-theorem into simpler forms, which is also a meta-theorem. The base case corresponds to the meta-theorems that can be easily proved, for example, a logical tautology.

The induction framework for automatically proving the subject reduction lemma can be easily extended to check other properties.

The SL system is available from:

<http://www.cs.uoregon.edu/~ariola/SL/>.

### Organization

This thesis is organized as follows: Chapter II demonstrates how the SL system generates interpreters from syntactic theories; Chapter III describes how to automatically prove the unique-decomposition lemma by using and extending tree automata al-

gorithms; Chapter IV extends the previous SL system to allow abstract patterns, natural deduction specifications, and internal substitutions; Chapter V presents the meta-level reasoning used to automatically prove subject reduction; Chapter VI summarizes and concludes.

## CHAPTER II

### THE SL SYSTEM – GENERATING INTERPRETERS FROM SYNTACTIC THEORIES

In this chapter, we introduce a specification language, SL, which is tailored to the writing of syntactic theories of language semantics. More specifically, the language supports the specification of primitive notions such as dynamic constraints, contexts, axioms, and inference rules. We also introduce a system which generates interpreters from SL specifications. A prototype system is implemented and has been tested on a number of examples, including a syntactic theory for Verilog.

We first provide an overview of the SL system by using the call-by-value  $\lambda$ -calculus as an example. Then we describe the constructs of the SL language, demonstrate its compilation, and discuss the runtime library for the SL system. We also show an example of a generated interpreter before our closing summary and discussion.

#### Overview of the SL System

We use the call-by-value  $\lambda$ -calculus [Plo75] as an example and show how it is specified in the SL language. By convention, we call the SL language the *meta-language*, and call the call-by-value  $\lambda$ -calculus the *object-language*.

#### Representation in SL

The specification of the call-by-value  $\lambda$ -calculus in SL is given in Figures 4. The SIGNATURE part describes the abstract syntax of the language using CAML type def-

---

```

SIGNATURE:
type M = Var of string | Lam of string*M | App of M*M;;
startfrom M;;

SPECIFICATION:
#open "namesupply";;
let rec subst (t1,x,t2) =
  match t1 with
  | Var s -> if s = x then t2 else t1
  | Lam(s,t1') -> if s = x then t1
    else let s' = freshname() in
      Lam(s', subst(subst(t1',s,Var s'),x,t2))
  | App(t11,t12) -> App(subst(t11,x,t2),subst(t12,x,t2));;

dynamic V = Lam _;;

axiom betav: App(Lam(x,t1), (t2:V)) ==> subst(t1, x, t2);;

context H = BOX | App(H,_) | App(V,H);;

inference eval:
t1 ==> t2
-----
(h:H) t1 ==> h t2 ;;

```

---

FIGURE 4. An SL Specification of a Simple CBV Language

initions. In general, the SL type definitions may be polymorphic but are restricted to first-order type expressions, *i.e.*, expressions that do not contain function types. To account for cases in which the description needs more than one type, the type of programs in the object-language is explicitly given by the `start from` phrase.

The SPECIFICATION part describes the semantics of the language. A dynamic definition defines a subset of a type with some semantic significance. The meta-language also has a primitive notion of contexts with `BOX` as the empty context. The axioms are conditional rewriting rules. Each inference rule has one premise clause, one conclusion clause, and an optional condition expression. The optional conditions are CAML expressions following the keyword `when`. Axioms and inference rules use a richer notion of pattern-matching than the one used in most functional languages. They include dynamic constraints as in  $\tau_2 : V$ , context constraints as in  $h : H$ , and context fillings such as  $h \ \tau_2$ . Meta-operations of the semantics like substitution are written directly in CAML as auxiliary definitions.

The BNF of the SL language is given in Appendix A.

### Generating interpreters

The SL system is *very* domain-specific, targeting exactly the kind of semantic specifications based on syntactic theories. In addition, it performs basic checks to ensure the specifications are well-formed. Other than the basic syntactic checks, the SL system has a (meta-)type system that ensures that contexts are used appropriately, *e.g.*, every context has one hole, contexts are filled with expressions of the appropriate types, and both sides of each axiom have the same type. After performing these basic checks, the SL system compiles the specification into a non-deterministic automaton, which is



then transliterated into CAML code. The code uses success continuations to encode the sequencing of states and uses exceptions with handlers to encode the non-deterministic selection of a state.

Feeding the code in Figure 4 to the SL system produces an interpreter. This interpreter can then be invoked on terms of the language to evaluate them by repeatedly decomposing them into evaluation contexts and redexes, and contracting the redexes, until an answer is reached. For example, if the input file contains:

---

```
App (Lam ("y", Var "y"), App (Lam ("x", Var "x"), Lam ("z", Var "z"))) ; ;
```

---

the generated interpreter produces:

---

```
App (Lam ("y", Var "y"), App (Lam ("x", Var "x"), Lam ("z", Var "z")))
==>   by betav, eval
App (Lam ("y", Var "y"), Lam ("z", Var "z"))
==>   by betav, eval
Lam ("z", Var "z")
```

---

Interpreters generated by the SL system preserve the semantics of an object-language in the sense that if a specification is non-deterministic, the generated interpreter evaluates input programs non-deterministically.

### The SL Language

Because CAML phrases can be used in SL specifications, the SL language contains all the constructs of CAML such as expressions, patterns, and type expressions. In addition, the SL language has constructs specific to the SL system, such as dynamic expressions, context expressions, and their corresponding declarations.

The abstract syntax of the SL language is shown in Figure 5. We write  $x$  for variables,  $c_0$  for nullary constructors,  $c_1$  for constructors of arity one, *const* for constant expressions,  $\alpha$  for type variables,  $c_T$  for built-in types, and *op* for built-in operations. Note that the symbol “=” and the symbol “|” in the figure are symbols of the SL language. We will explain type expressions, expressions, patterns, declarations and definitions, axioms, and inference rules next.

#### 1. Type Expressions

Type expressions in the SL language are similar to those in the CAML language. The only difference is that the SL types need to be first-order when used to specify the datatypes of the object-language. The types allowed for object-datatypes can be type variables, constant built-in types such as `bool` and `int`, a type name already defined, tuples, and lists. Type variables make the type system in the SL language polymorphic.

#### 2. Expressions

SL Expressions are also similar to the expressions in CAML. The expressions can be one of the following forms: variables, constants, constructors, built-in operations, type constraint expressions, functions, applications, context filling, let expressions, or letrec expressions.

<i>Type Expressions</i>	$T_E ::= \alpha \mid c_T \mid \text{type\_name} \mid T_E * \dots * T_E \mid T_E \text{ list}$
<i>Expressions</i>	$E ::= x \mid \text{const} \mid c_0 \mid c_1 E \mid (E, \dots, E) \mid \text{op } E \mid \lambda x. E \mid E E \mid \text{let } x = E \text{ in } E \mid \text{letrec } x = E \text{ in } E \mid (E : \text{type } T_E) \mid E [E]$
<i>Patterns</i>	$P ::= - \mid x \mid \text{const} \mid c_0 \mid c_1 P \mid P P \mid P \text{ as } x \mid (P : \text{type } T_E) \mid (P, \dots, P) \mid (P : \text{dynamic\_name}) \mid \square \mid (P : \text{context\_name}) [P]$
<i>Type Declaration</i>	$TDecl ::= \text{type\_name} = c_0 \mid \dots \mid c_1 T_E$
<i>Dynamic Definitions</i>	$DDecl ::= \text{dynamic\_name} = P$
<i>Context Definitions</i>	$CDecl ::= \text{context\_name} = P \mid \dots \mid P$
<i>Axioms</i>	$A ::= ax\_name : P \text{ when } E ==> E$
<i>Inference Rules</i>	$I ::= inf\_name : \frac{E ==> P}{P \text{ when } E ==> E}$

FIGURE 5. Abstract syntax of the SL language

Other forms such as exceptions and memory references are supported in the SL language. They are excluded in this thesis because they are not essential to specify the object-language constructs. Also, let expressions and letrec expressions are simplified to have only one binding clause.

### 3. Patterns

SL patterns include common patterns such as wildcard patterns, variable patterns, constant patterns, alternative patterns, type-constraint patterns, alias patterns, and tuple patterns. The SL patterns are also enriched with dynamic constraint patterns and context filling patterns. The dynamic constraint pattern  $(p : \textit{dynamic\_name})$  and context filling pattern  $(p : \textit{context\_name}) [p_2]$  require  $p$  to be a wildcard pattern or a variable pattern.

SL patterns can be used in the left-hand sides of axioms and the left-hand sides of conclusion clauses in the inference rules. They can also be used in the right-hand side of premise clauses of inference rules, but those patterns should not contain dynamic constraints or context fillings.

*Context expressions* are special kinds of patterns. Each context expression has a hole  $\square$  that is not filled with a pattern. For example,  $\square$ ,  $c_1(\square)$ ,  $\textit{context\_name}$ , and  $(p : \textit{context\_name}) [\square]$  are context expressions, while  $c_0$ ,  $\textit{const}$ , and  $(p : \textit{context\_name}) [c_0]$  are not. Context expressions cannot be top-level patterns except in context declarations.

### 4. Declarations and Definitions

Datatype declarations in the SL language are similar to those in CAML. The difference is that they can use only first-order type expressions. Type declarations

introduce new constructors as well as type names.

Dynamic definitions can be considered as definitions of SL patterns, and context definitions are parametric patterns where the parameters denote the holes. Each alternative pattern in the right-hand side of a context definition must be a context expression. Dynamic definitions and context definitions can be recursively defined.

## 5. Axioms

Each axiom has one name for easy reference. Its left-hand-side pattern should be linear. In other words, no variable can occur more than once in the pattern. The right-hand side is an expression and the condition is also an expression. The set of free variables in the two expressions is restricted to be a subset of the free variables in left-hand pattern.

## 6. Inference Rules

Each inference rule has a name, a premise clause, and a conclusion clause. The premise clause has an expression as its left-hand side and a pattern as its right-hand side, whereas the conclusion clause has a pattern as its left-hand side and an expression as its right-hand side.

Similarly to the restrictions for axioms, both patterns in the premise clause and the conclusion clause should be linear. The set of free variables on the right-hand side of the conclusion clause should be a subset of the union of free variables on the left and free variables in premise clauses. Moreover, the free variables of the left-hand side expression of a premise clause should be a subset of the free variables in the corresponding left-hand pattern of the conclusion clause.

The dynamic definition, context definition, and rules in Figure 4 can be represented in the abstract syntax of the SL language as follows:

$$\begin{aligned}
 V &= \text{Lam } \_ \\
 H &= \square \mid \text{App}((h_1 : H)(\square), \_) \mid \text{App}((v : V), (h_2 : H)(\square)) \\
 \beta_v &: \text{App}(\text{Lam}(x, M), (v : V)) \implies M[x := v] \\
 eval &: \frac{t_1 \implies t_2}{(h : H) t_1 \implies h t_2}
 \end{aligned}$$

where variables are introduced for dynamic constraint and context filling patterns.

### Compiling SL specifications

The compilation of an SL specification includes the usual phases such as lexing, parsing, static checking, and code generation. For an object-language specified by an SL program, a parser and a pretty-printer for the object-language are generated from the signature part, and a reduction machine based on pattern-matching automata is generated from the semantic rules. These parts work together as an interpreter for the object-language with the support of the SL runtime libraries.

In this section, we address some special issues in the SL compilation such as type checking, pattern matching and building automata, and transforming automata into CAML code.

### Type System

The type system for SL extends the type system for CAML. The extensions deal with dynamic definitions and contexts. Here, we only present the idea of typing expressions and typing patterns in a simply-typed framework.

$\overline{\Gamma \vdash \alpha : \tau}$	$\overline{\Gamma, c_T : c_T \vdash c_T : c_T}$
$\overline{\Gamma, tn : \tau \vdash tn : \tau}$	$\frac{\Gamma \vdash te_1 : \tau_1 \quad \dots \quad \Gamma \vdash te_n : \tau_n}{\Gamma \vdash te_1 * \dots * te_n : \tau_1 * \dots * \tau_n}$
$\frac{\Gamma \vdash te : \tau}{\Gamma \vdash te list : \tau list}$	

TABLE 1. The type checking rules for type expressions

First, we give definitions of types and context types in the SL type system.

$$\begin{aligned} \text{Types} \quad T &::= \alpha \mid c_T \mid T * \dots * T \mid T list \mid T \rightarrow T \\ \text{Context Types} \quad U &::= T \circ \rightarrow T \end{aligned}$$

A type can be either a type variable, built-in type, tuple type, list type, or function type. Note that although type expressions in the SL language do not have function types, the types in the SL type system can. A context type has the form of  $t_1 \circ \rightarrow t_2$ , where  $t_1$  is the type for the hole and  $t_2$  is the type of the whole expression if the hole is filled. Only context expressions and context definitions have context type.

In the following, we discuss the type checking rules for the SL constructs.

### 1. Typing type expressions

The typing rules for type expressions are given in Table 1. The rules are straightforward.

### 2. Typing expressions

$\overline{\Gamma, x : \tau \vdash x : \tau}$	$\overline{\Gamma, const : c_T \vdash const : c_T}$
$\overline{\Gamma, c_0 : \tau \vdash c_0 : \tau}$	$\overline{\Gamma, c_1 : \tau_1 \rightarrow \tau_2 \vdash c_1 : \tau_1 \rightarrow \tau_2}$
$\frac{\Gamma \vdash c_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash c_1 e : \tau_2}$	$\frac{\Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash (e_1, \dots, e_n) : \tau_1 * \dots * \tau_n}$
$\overline{\Gamma, op : \tau_1 \rightarrow \tau_2 \vdash op : \tau_1 \rightarrow \tau_2}$	$\frac{\Gamma \vdash op : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash op e : \tau_2}$
$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2}$	$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$
$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e : \tau}$	$\frac{\Gamma, x : \tau_1 \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e : \tau}{\Gamma \vdash \text{letrec } x = e_1 \text{ in } e : \tau}$
$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash T_E : \tau}{\Gamma \vdash (e : \text{type } T_E) : \tau}$	
$\overline{\Gamma, x : \tau_1 \circ \rightarrow \tau_2 \vdash x : \tau_1 \circ \rightarrow \tau_2}$	$\frac{\Gamma \vdash e_1 : \tau_1 \circ \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1[e_2] : \tau_2}$

TABLE 2. The type checking rules for SL expressions



The typing rules for SL expressions are given in Table 2. The first part of the table is the set of rules for expressions except context filling. Those rules are standard. The second part of the table is the rule for context-filling expressions, which is similar to the rule for function applications. For instance, if  $e_1$  in a context filling expression  $e_1[e_2]$  has type  $\tau_1 \circ \rightarrow \tau_2$  and  $e_2$  has type  $\tau_1$ , then the resulting type of  $e_1[e_2]$  is  $\tau_2$ .

### 3. Typing patterns

The typing rules for SL patterns are given in Table 3. The first part of the table is the set of rules for common patterns. Those rules are standard as well. The second part is the set of rules for context expressions and context-constraint patterns. The typing rule for the  $\square$  pattern gives the pattern the type  $\tau \circ \rightarrow \tau$ . The rule for context-constraint patterns is similar to the rule for function applications if the patterns to be filled do not have context type. Typing rules for other context expressions express the “lifting” of the context type constructor  $\circ \rightarrow$  whenever possible, so that context expressions preserve context types. If a pattern  $p_1$  has context type  $\tau_0 \circ \rightarrow \tau_1$  and another pattern  $p_2$  has type  $\tau_2$ , then the pair pattern  $(p_1, p_2)$  has type  $(\tau_0 \circ \rightarrow (\tau_1 * \tau_2))$  instead of  $((\tau_0 \circ \rightarrow \tau_1) * \tau_2)$ . The context type constructor  $\circ \rightarrow$  is always at the top level of a type. Another example is the rule for context constraint patterns if the patterns to be filled are context types. If a context  $cn$  has type  $\tau_1 \circ \rightarrow \tau_2$  and a context expression  $p_2$  has type  $\tau_0 \circ \rightarrow \tau_1$ , then the context expression  $(p_1 : cn)[p_2]$  has type  $\tau_0 \circ \rightarrow \tau_2$  instead of  $(\tau_0 \circ \rightarrow \tau_1) \circ \rightarrow \tau_2$ . Such behavior is like function composition for context types.

### 4. Typing declarations

$\overline{\Gamma \vdash \_ : \tau}$	$\overline{\Gamma, x : \tau \vdash x : \tau}$
$\overline{\Gamma, const : \tau \vdash const : \tau}$	$\overline{\Gamma, c_0 : \tau \vdash c_0 : \tau}$
$\frac{\Gamma \vdash p : \tau_1}{\Gamma, c_1 : \tau_1 \rightarrow \tau_2 \vdash c_1 p : \tau_2}$	$\frac{\Gamma \vdash p_1 : \tau_1 \quad \dots \quad \Gamma \vdash p_n : \tau_n}{\Gamma \vdash (p_1, \dots, p_n) : \tau_1 * \dots * \tau_n}$
$\frac{\Gamma \vdash p : \tau}{\Gamma, x : \tau \vdash p \text{ as } x : \tau}$	$\frac{\Gamma \vdash p_1 : \tau \quad \Gamma \vdash p_2 : \tau}{\Gamma \vdash p_1   p_2 : \tau}$
$\frac{\Gamma \vdash p : \tau \quad \Gamma \vdash T_E : \tau}{\Gamma \vdash (p : \text{type } T_E) : \tau}$	$\frac{\Gamma \vdash p : \tau}{\Gamma, dn : \tau \vdash (p : dn) : \tau}$
<hr/>	
$\overline{\Gamma \vdash \square : \tau_0 \rightarrow \tau}$	$\frac{\Gamma \vdash p_1 : \tau_0 \rightarrow \tau_1 \quad \Gamma \vdash p_2 : \tau_0}{\Gamma, cn : \tau_0 \rightarrow \tau_1 \vdash (p_1 : cn)[p_2] : \tau_1}$
$\frac{\Gamma \vdash c_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash p : \tau_0 \rightarrow \tau_1}{\Gamma \vdash c_1 p : \tau_0 \rightarrow \tau_2}$	$\frac{\Gamma \vdash p_1 : \tau_1 \quad \Gamma \vdash p_i : \tau_0 \rightarrow \tau_i \quad \Gamma \vdash p_n : \tau_n}{\Gamma \vdash (p_1, \dots, p_i, \dots, p_n) : \tau_0 \rightarrow \tau_1 * \dots * \tau_n}$
$\frac{\Gamma \vdash p_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash p_2 : \tau_0 \rightarrow \tau_1}{\Gamma, cn : \tau_1 \rightarrow \tau_2 \vdash (p_1 : cn)[p_2] : \tau_0 \rightarrow \tau_2}$	

TABLE 3. The type checking rules for SL patterns

<p><i>Type Declaration:</i></p> $\Gamma, tn : \tau, c_0 : \tau, \dots, c_1 : \sigma \rightarrow \tau \vdash c_0 : \tau$ $\vdots$ $\Gamma, tn : \tau, c_0 : \tau, \dots, c_1 : \sigma \rightarrow \tau \vdash te : \sigma$ <hr style="border: 0.5px solid black;"/> $\Gamma, tn : \tau, c_0 : \tau, \dots, c_1 : \sigma \rightarrow \tau \vdash tn = c_0 \mid \dots \mid c_1 te : \tau$
<p><i>Dynamic Definitions:</i></p> $\Gamma, dn : \tau \vdash p_1 : \tau$ $\vdots$ $\Gamma, dn : \tau \vdash p_n : \tau$ <hr style="border: 0.5px solid black;"/> $\Gamma, dn : \tau \vdash dn = p_1 \mid \dots \mid p_n : \tau$
<p><i>Context Definitions:</i></p> $\Gamma, cn : \sigma \rightarrow \tau \vdash p_1 : \sigma \rightarrow \tau$ $\vdots$ $\Gamma, cn : \sigma \rightarrow \tau \vdash p_n : \sigma \rightarrow \tau$ <hr style="border: 0.5px solid black;"/> $\Gamma, cn : \sigma \rightarrow \tau \vdash cn = p_1 \mid \dots \mid p_n : \sigma \rightarrow \tau$

TABLE 4. The type checking rules for declarations

$\Gamma, x_1 : \tau_1, \dots, x_n : \tau_n \vdash p : \tau$ $\Gamma, x_1 : \tau_1, \dots, x_n : \tau_n \vdash e_c : bool$ $\frac{\Gamma, x_1 : \tau_1, \dots, x_n : \tau_n \vdash e : \tau}{\Gamma \vdash ax\_name : p \text{ when } e_c ==> e : \tau}$
---

TABLE 5. The type checking rules for axioms

The typing rules for type declarations, dynamic definitions, and context definitions are given in Table 4.

For a type declaration, all the alternatives should have the same type and the whole declaration is considered to have this type as well. Newly introduced constructors and the type name are added into the typing environment.

For a dynamic definition or a context definition, all the alternative patterns should have the same type and the whole definition will have this type as well. This type cannot be a context type in the case of a dynamic definition, however it must be a context type in the case of a context definition. The dynamic name or context name is added into the typing environment.

### 5. Typing axioms

The typing rule for axioms is given in Table 5. Both sides of an axiom should have the same type and the type cannot be a context type. The axiom is considered to have that type as well. The condition expression in an axiom should have boolean type.

$\Gamma, x_1 : \tau_1, \dots, x_n : \tau_n \vdash p : \tau$
$\Gamma, x_1 : \tau_1, \dots, x_n : \tau_n \vdash e_c : bool$
$\Gamma, x_1 : \tau_1, \dots, x_n : \tau_n \vdash e_0 : \sigma$
$\Gamma, x_1 : \tau_1, \dots, x_n : \tau_n, y_1 : \sigma_1, \dots, y_m : \sigma_m \vdash p_0 : \sigma$
$\Gamma, x_1 : \tau_1, \dots, x_n : \tau_n, y_1 : \sigma_1, \dots, y_m : \sigma_m \vdash e : \tau$
$\Gamma \vdash inf\_name : \frac{e_0 ==> p_0}{p \text{ when } e_c ==> c} : \tau$

TABLE 6. The type checking rules for inference rules

## 6. Typing inference rules

The typing rule for inference rules is given in Table 6. Similarly to the rule for axioms, both sides of either clause in an inference rule should have the same type. The type of the premise clause and the type of the conclusion clause are not necessarily the same, but neither of the types can be context types. The type of the whole inference rule is the same as the type of its conclusion clause. The condition expression in an inference should have boolean type.

### Building pattern-matching automata

Pattern matching is the crucial part in compiling an SL specification. A naive way is to check a list of patterns one by one. The obvious drawback of this approach is inefficiency. Tree-like automata [GS84] address the efficiency issue. Matching proceeds by making branches of different constructors and ascribing the list of patterns to those

branches. This approach has the disadvantage of space explosion. The combination of tree automata with failures is the basis for the current implementations of most common functional languages [Mar94, Ler90]. Pattern matching in the SL system follows this approach, but it requires extensions to the existing algorithms to support our semantic notions and the non-determinism of rewriting.

### 1. Automata

An automaton consists of states. Some states are final. Matching a term consists of traversing the states until reaching a final one. States are inductively defined as follows:

$$\begin{aligned} \text{States } S ::= & \text{branch } (t, (test, S), \dots, (test, S)) \mid \text{accept } E \mid \\ & S \mid \dots \mid S \mid \text{fail} \mid \text{let } vars = f E \text{ in } S \mid \\ & \text{if } E \text{ then } S \text{ else } S \mid \text{let } vars = vars \text{ in } S \end{aligned}$$

where  $vars$  denotes a variable or a tuple of variables.

The first four states are standard.  $\text{branch } (t, (test_1, S_1), \dots, (test_n, S_n))$  is a branch-test state, where  $t$  is the term under test, and each  $test_i$  has the form  $c_0$ ,  $c_1 x$ , or “\_” for otherwise. All tests are mutually disjoint.  $\text{accept } e$  is a final state, where  $e$  is a CAML expression representing the action after acceptance. A choice state  $S_1 \mid \dots \mid S_n$  has alternatives  $S_1, \dots, S_n$ . When pattern matching traverses this state, it non-deterministically chooses to enter an alternative. If a final state is reached, then traversing the choice state is successful. Otherwise it backtracks to other alternatives. This semantics differs from the usual choice state whose

alternatives are ordered (lexically). **fail** is a failure state. One new form, reference state  $\text{let } vars = f e \text{ in } S_1$ , is added to support dynamic values and contexts. It calls a function matching the parameter as the corresponding dynamic value, context, or  $\text{redex}$ . If it succeeds, it continues in state  $S_1$ . Failures in  $S_1$  may cause backtracking to other possibilities in the function call  $f e$ .

States are also extended with conditional expressions and let variable bindings. The reason for the former extension is that the semantic rules are conditional. The latter extension is helpful for code generation. States are annotated with terms to be matched, but we made them implicit in our presentation.

## 2. Structures for pattern-matching

The SL compiler collects the inference rules and axioms with the same type together. The patterns of the rules form a vector which can be considered as a one-column matrix. Each rule contributes a row in the matrix. We introduce parameters bound to the terms matching the patterns, and we can keep track of the variable bindings in the pattern matching. There is also a state for each rule, indicating what to do when the patterns of the rule have been matched. The whole pattern-matching structure is represented as follows:

$$\begin{pmatrix} t_1 \cdots t_n \\ p_{11} \cdots p_{1n} & s_1 \\ \vdots \cdots \vdots & \vdots \\ p_{m1} \cdots p_{mn} & s_m \end{pmatrix}$$

The compilation of pattern-matching can be regarded as a function,  $\mathcal{C}$ , which maps such a structure to a state.

The initialization of the pattern matching sets the states in the structure.

- For an axiom  $p_1$  when  $e_c \implies e_r$ , the corresponding state is:

**if  $e_c$  then accept  $e_r$  else fail**

- For an inference rule  $\frac{e_1 \implies p_1}{p_2 \text{ when } e_c \implies e_2}$ , the corresponding state is:

**if  $e_c$  then let  $p_1 = \text{rewrite1 } e_1$  in accept  $e_2$  else fail**

where **rewrite1** is one-step rewriting function in the generated code.

### 3. Pattern-matching algorithm

The pattern-matching algorithm is a divide-and-conquer algorithm. It selects one column of the pattern matrix to work on according to certain criteria. Without loss of generality, we assume that the algorithm always chooses the first column. The algorithm repeats the following steps until the pattern matrix is empty:

#### (a) Preprocessing

This step canonicalizes the patterns in the first column. It removes the type constraints since the type information is not useful at the current stage. It binds variables in alias patterns to the corresponding parameters. It turns each alternative pattern into several rows having one alternative each. Formally, the preprocessing repeats the following simplifications.



$$\begin{array}{ccc}
\mathcal{C} \begin{pmatrix} t_1 & \dots \\ \vdots & \dots \vdots \\ (p_{i1} : \text{type } \tau) \cdots s_i \\ \vdots & \dots \vdots \end{pmatrix} & \longrightarrow & \mathcal{C} \begin{pmatrix} t_1 \cdots \\ \vdots \cdots \vdots \\ p_{i1} \cdots s_i \\ \vdots \cdots \vdots \end{pmatrix} \\
\mathcal{C} \begin{pmatrix} t_1 & \dots \\ \vdots & \dots \vdots \\ (p_{i1} \text{ as } x) \cdots s_i \\ \vdots & \dots \vdots \end{pmatrix} & \longrightarrow & \mathcal{C} \begin{pmatrix} t_1 \cdots & \\ \vdots \cdots & \vdots \\ p_{i1} \cdots \text{let } x = t_1 \text{ in } s_i \\ \vdots \cdots & \vdots \end{pmatrix} \\
\mathcal{C} \begin{pmatrix} t_1 & \dots \\ \vdots & \dots \vdots \\ p_{i1a} | p_{i1b} \cdots s_i \\ \vdots & \dots \vdots \end{pmatrix} & \longrightarrow & \mathcal{C} \begin{pmatrix} t_1 \cdots \\ \vdots \cdots \vdots \\ p_{i1a} \cdots s_i \\ p_{i1b} \cdots s_i \\ \vdots \cdots \vdots \end{pmatrix}
\end{array}$$

## (b) Splitting the matrix

The algorithm splits the matrix horizontally so that in each submatrix, all first-column patterns are in one of the following groups:

- *variable group*: wildcard patterns or variable patterns,
- *tuple group*: tuple patterns,
- *constructor group*: constructor patterns,
- *dynamic constraint group*: dynamic constraint patterns on the same dynamic definition,
- *context filling group*: context filling patterns on the same context defini-

tion.

The result of the splitting is a choice state.

$$\mathcal{C} \begin{pmatrix} t_1 \cdots \\ p_{11} \cdots s_1 \\ \vdots \cdots \vdots \\ p_{m1} \cdots s_m \end{pmatrix} \rightarrow \mathcal{C} \begin{pmatrix} t_1 \cdots \\ p_{11} \cdots s_1 \\ \vdots \cdots \vdots \\ p_{k_1 1} \cdots s_{k_1} \end{pmatrix} \mid \cdots \mid \mathcal{C} \begin{pmatrix} t_1 \cdots \\ p_{(k_l+1)1} \cdots s_{k_l+1} \\ \vdots \cdots \vdots \\ p_{m1} \cdots s_m \end{pmatrix}$$

(c) Analyzing different cases

For each submatrix, the compilation function  $\mathcal{C}$  is inductively defined as follows:

i. Base case:

When the pattern matrix is empty, pattern-matching is vacuously successful. The function  $\mathcal{C}$  creates a choice state.

$$\mathcal{C} \begin{pmatrix} s_1 \\ \vdots \\ s_m \end{pmatrix} \rightarrow s_1 \mid \cdots \mid s_m$$

ii. Variable group:

Wildcard and variable patterns always match successfully. The function  $\mathcal{C}$  removes the column of patterns. For variable patterns, bindings are added for further access to the variables.

$$\mathcal{C} \begin{pmatrix} t_1 \cdots \\ \vdots \cdots \vdots \\ \cdots s_i \\ \vdots \cdots \vdots \\ x \cdots s_j \\ \vdots \cdots \vdots \end{pmatrix} \rightarrow \mathcal{C} \begin{pmatrix} t_2 \cdots \\ \vdots \cdots \vdots \\ p_{i2} \cdots s_i \\ \vdots \cdots \vdots \\ p_{j2} \cdots \text{let } x = t_1 \text{ in } s_j \\ \vdots \cdots \vdots \end{pmatrix}$$

iii. Tuple group:

The function  $\mathcal{C}$  treats each component of the tuple as an individual pattern. It replaces the first column of patterns with columns of the component patterns, and introduce parameters for the components.

$$\mathcal{C} \begin{pmatrix} t_1 & \cdots \\ (p_{111}, \cdots, p_{11k}) \cdots s_1 \\ \vdots & \cdots \vdots \\ (p_{m11}, \cdots, p_{m1k}) \cdots s_m \end{pmatrix} \rightarrow \begin{matrix} \text{let } (t_{11}, \cdots, t_{1k}) = t_1 \text{ in} \\ \mathcal{C} \begin{pmatrix} t_{11} \cdots t_{1k} \cdots \\ p_{111} \cdots p_{11k} \cdots s_1 \\ \vdots \cdots \vdots \cdots \vdots \\ p_{m11} \cdots p_{m1k} \cdots s_m \end{pmatrix} \end{matrix}$$

iv. Constructor group:

The function  $\mathcal{C}$  collects the rows which have the same constructor in the first column into a group of new pattern-matching structures, and it creates a branch-test state. The tests of the state are distinguished by having different constructors as roots. The corresponding actions for

the tests are the results of compiling the new structures. In the new structures, the first column is removed for the constructors of zero arity, or it is replaced by the column of argument patterns for the constructors of non-zero arity.

$$\mathcal{C} \begin{pmatrix} t_1 & \cdots \\ \vdots & \cdots \vdots \\ c_0 & \cdots s_i \\ \vdots & \cdots \vdots \\ c_1 p'_{j_1} \cdots s_j \\ \vdots & \cdots \vdots \end{pmatrix} \rightarrow \text{branch} \left( t_1, \begin{pmatrix} (c_0, \mathcal{C} \begin{pmatrix} t_2 & \cdots \\ p_{i_1 2} \cdots s_{i_1} \\ \vdots & \cdots \vdots \\ p_{i_2 2} \cdots s_{i_2} \end{pmatrix}) \\ \vdots \\ \text{let } c_1 t'_1 = t_1 \text{ in} \\ (c_1 t'_1, \mathcal{C} \begin{pmatrix} t'_1 & \cdots \\ p'_{j_1 1} \cdots s_{j_1} \\ \vdots & \cdots \vdots \\ p'_{j_2 1} \cdots s_{j_2} \end{pmatrix}) \end{pmatrix} \right)$$

v. Dynamic constraint group:

The function  $\mathcal{C}$  creates a reference state. The reference will initiate pattern matching for the dynamic definition  $D$  with the value of  $t_1$ . Its result is bound to a new parameter. The state in the let body is the result of compiling a structure consisting of the patterns without the constraint and the rest of the patterns.

$$\begin{array}{c}
\mathcal{C} \left( \begin{array}{ccc} t_1 & \cdots & \\ (p'_{11} : D) \cdots s_1 & & \\ \vdots & \cdots & \vdots \\ (p'_{m1} : D) \cdots s_m & & \end{array} \right) \longrightarrow \text{let } t'_1 = \text{match\_}D \ t_1 \text{ in} \\
\mathcal{C} \left( \begin{array}{ccc} t'_1 & \cdots & \\ p'_{11} \cdots s_1 & & \\ \vdots & \cdots & \vdots \\ p'_{m1} \cdots s_m & & \end{array} \right)
\end{array}$$

Pattern matching for dynamic definitions will be presented later.

vi. Context filling group

Similar to the dynamic constraint group, the function  $\mathcal{C}$  creates a reference state. The reference will initiate pattern-matching for the context definition  $H$  with the value of  $t_1$ . Its result is bound to a pair of parameters which represent the context and the corresponding hole occurring in  $t_1$ . The state of the let body is the result of compiling a structure consisting of the context patterns, the hole patterns, as well as the rest of the patterns.

$$\begin{array}{c}
\mathcal{C} \left( \begin{array}{ccc} t_1 & \cdots & \\ (p'_{11} : H)p''_{11} \cdots s_1 & & \\ \vdots & \cdots & \vdots \\ (p'_{m1} : H)p''_{m1} \cdots s_m & & \end{array} \right) \longrightarrow \text{let } (t'_1, t''_1) = \text{match\_}H \ t_1 \text{ in} \\
\mathcal{C} \left( \begin{array}{ccc} t'_1 & t''_1 & \cdots \\ p'_{11} & p''_{11} \cdots s_1 & \\ \vdots & \vdots & \cdots \vdots \\ p'_{m1} & p''_{m1} \cdots s_m & \end{array} \right)
\end{array}$$

Pattern matching for context definitions will be presented next.

#### 4. Matching dynamic definitions and context definitions

Pattern-matching for dynamic definitions and context definitions uses the same form of structures and uses the same algorithm. Each definition corresponds to a structure which has only one pattern, one parameter, and one state. The pattern comes from the definition. Assuming  $t$  is the term to be matched, the state is initialized as follows:

- For a dynamic definition, the state is **accept**  $t$ .
- For a context definition, the state is **accept**  $(\lambda x. body, x)$ , where  $x$  is the parameter for the context and the *body* is a placeholder. When the base case in the algorithm is encountered, the SL compiler sets the contents of *body* by reconstructing the term  $t$  with the variable  $x$ . Each *body* may have different content if the final state is copied. The reconstruction retrieves the bindings along the path from the start state to the final state.

In other words, matching a context definition returns a pair with a constructing function and a term filling the hole. Applying the function to the term results in the term  $t$ .

The state starting pattern matching for a definition can be referred to by the name of the definition. For example, the dynamic definition and the context definition

in Figure 4 are associated with the following states:

$$\text{match}_D t = \text{branch}(t, (\text{Lam } t', \text{accept}(t)))$$

$$\begin{aligned} \text{match}_H t = & \text{let } x = t \text{ in } \text{accept}(\lambda x. \text{body}_1, x) \quad | \\ & \text{branch}(t, \\ & \quad (\text{App } t', \text{let } (t_1, t_2) = t' \text{ in} \\ & \quad \quad (\text{let } (t'_1, t''_1) = \text{match}_H t_1 \text{ in} \\ & \quad \quad \quad \text{let } h_1 = t'_1 \text{ in} \\ & \quad \quad \quad \quad \text{let } x = t''_1 \text{ in } \text{accept}(\lambda x. \text{body}_2, x)) \quad | \\ & \quad \quad (\text{let } t'_1 = \text{match}_D t_1 \text{ in let } v = t'_1 \text{ in} \\ & \quad \quad \quad \text{let } (t'_2, t''_2) = \text{match}_H t_2 \text{ in} \\ & \quad \quad \quad \quad \text{let } h_2 = t'_2 \text{ in} \\ & \quad \quad \quad \quad \quad \text{let } x = t''_2 \text{ in } \text{accept}(\lambda x. \text{body}_3, x)))) \end{aligned}$$

where  $\text{body}_1$ ,  $\text{body}_2$  and  $\text{body}_3$  are as follows:

$$\text{body}_1 : \text{let } t = x \text{ in } t$$

$$\text{body}_2 : \text{let } t'_1 = x \text{ in let } t'_1 = h_1 \text{ in let } t_1 = t'_1 \text{ in } t'_1$$

$$\text{let } t' = (t_1, t_2) \text{ in let } t = \text{App } t' \text{ in } t$$

$$\text{body}_3 : \text{let } t'_2 = x \text{ in let } t'_2 = h_2 \text{ in let } t_2 = t'_2 \text{ in } t'_2$$

$$\text{let } t'_1 = v \text{ in let } t_1 = t'_1 \text{ in let } t' = (t_1, t_2) \text{ in } t$$

$$\text{let } t = \text{App } t' \text{ in } t$$

### Transforming automata into CAML code

The transformation of states in the pattern-matching automata into CAML code is described in Figure 6. Each state corresponds to a function with implicit parameters for the terms to be matched and a continuation expressing the remaining pattern-matching work. The initial continuation for rewriting is the identity function. State functions may raise an exception when matching for the state fails. The branch-test state corresponds to the `match ... with ...` construct in CAML. If the branch tests do not cover all constructors of the same type, a default test associated with a failure state is added. For a final state, the success continuation is consumed by applying it to the expression. For a choice state, a random alternative is tried. Failure exceptions may be caught and then other alternatives can be tried. The failure state just raises a failure exception. For a reference state, it calls the matching function with the continuation accepting the result, then it continues with the state in the body of the `let`. The conditional expressions and variable bindings in states are considered atomic with respect to continuation passing and exception handling. Transforming a conditional state is thus transforms both branch states, and transforming a `let` state is transforms the `in` part.

### The SL Runtime Libraries

The code generated by SL is mainly a rewriting engine. It cannot be executed directly as an interpreter. Runtime libraries in the SL system provide necessary utilities such as for parsing input terms, invoking and controlling reduction, and printing traces of reduction.



```

[ branch  $(t, (c_0, S_0), \dots, (c_1 x, S_1))$  ]  $k =$ 
  match  $t$  with
     $c_0 \quad - > [ S_0 ] k$ 
       $\vdots$ 
     $c_1 x \quad - > [ S_1 ] k$ 
     $- \quad - > \text{raise failure}$ 
[ accept  $e$  ]  $k = k(e)$ 
[  $S_1 | \dots | S_m$  ]  $k =$ 
  try  $[ S_i ] k$   $0 \leq i \leq m$ 
  with failure  $- >$ 
     $[ S_1 | \dots | S_{1-i} | S_{i+1} | \dots | S_m ] k$ 
[ fail ]  $k = \text{raise failure}$ 
[ let  $vars = f e$  in  $S$  ]  $k =$ 
   $f e (\lambda t. \text{let } vars = t \text{ in } [ S ] k)$ 
[ if  $e_c$  then  $S_1$  else  $S_2$  ]  $k =$ 
  if  $e_c$  then  $[ S_1 ] k$  else  $[ S_2 ] k$ 
[ let  $vars_1 = vars_2$  in  $S$  ]  $k =$ 
  let  $vars_1 = vars_2$  in  $[ S ] k$ 

```

FIGURE 6. Generating CAML code

### Runtime terms

Runtime terms are the terms to be evaluated at runtime. They are inputs for the generated interpreters. Runtime terms and runtime phrases are defined as follows:

*Runtime Terms*  $R ::= id \mid const \mid (R, \dots, R) \mid [R; \dots, R] \mid C(R)$

*Runtime Phrases*  $O ::= R \mid \text{rewrite number } R$

Runtime terms include identifiers, constants, tuples, lists, and the constructor  $C$ .  $C$  is the only constructor for runtime terms. Note that runtime terms are simple and general. Therefore, there exist universal parsers to parse the terms for all object-languages. The parser scans inputs and builds runtime terms which correspond to parse trees in the object-language. For example, the term

```
App (Lam ("y", Var "y"),
      App (Lam ("x", Var "x"), Lam ("z", Var "z")));;
```

is parsed into runtime term

```
C (App, (C (Lam, ("y", C (Var, "y"))),
          C (App, (C (Lam, ("x", C (Var, "x"))),
                  C (Lam, ("z", C (Var, "z"))))))))
```

Besides a term, the interpreter input can also specify the maximum number of reduction steps in evaluation. If the number is  $-1$ , there is no limit on the number of steps in evaluation.

Because the rewrite engine works on the object terms with specific datatypes given in the specification, transformation between runtime terms and object terms are needed.

$$\begin{aligned}
\llbracket const \rrbracket &= const \\
\llbracket C(Var, rt) \rrbracket &= Var \llbracket rt \rrbracket \\
\llbracket C(Lam, rt) \rrbracket &= Lam \llbracket rt \rrbracket \\
\llbracket C(App, rt) \rrbracket &= App \llbracket rt \rrbracket \\
\llbracket (rt_1, rt_2) \rrbracket &= (\llbracket rt_1 \rrbracket, \llbracket rt_2 \rrbracket) \\
\llbracket const \rrbracket^{-1} &= const \\
\llbracket Var x \rrbracket^{-1} &= C(Var, \llbracket x \rrbracket^{-1}) \\
\llbracket Lam t \rrbracket^{-1} &= C(Lam, \llbracket t \rrbracket^{-1}) \\
\llbracket App t \rrbracket^{-1} &= C(App, \llbracket t \rrbracket^{-1}) \\
\llbracket (t_1, t_2) \rrbracket^{-1} &= (\llbracket t_1 \rrbracket^{-1}, \llbracket t_2 \rrbracket^{-1})
\end{aligned}$$

FIGURE 7. Transformation between runtime terms and lambda terms

The transformations between runtime terms and the lambda terms defined in Figure 4 is given in Figure 7. We use  $\llbracket \cdot \rrbracket$  for transformation from runtime terms to lambda terms and use  $\llbracket \cdot \rrbracket^{-1}$  for transformation from lambda terms to runtime terms. Such transformations are straightforward so that they can be generated according to the data-type definition.

The input runtime terms must be successfully transformed into object-programs that have the startfrom type specified in SL. Syntax errors in the runtime terms can be detected when the transformation is unsuccessful. For the transformation in Figure 7, the runtime term  $C(Lam, ("x", "y", "z"))$  will encounter problems during translation to lambda terms. The problem is that the type for  $Lam$  is not consistent with the SIGNATURE in the SL specification.

Not all errors can be detected. Most of the semantic errors are not detected by the transformations. Users of the SL system may need to specify more rules to enforce reporting semantic errors.

## Runtime rewrite controls

Runtime rewrite controls include the continuation function, failure exception, rewriting iterations, and choosing non-deterministic states.

### 1. Success continuation and failure exception

The default success continuation is the identity function. It is used at the top level of rewriting, which is the entry point of the corresponding pattern-matching functions. The top-level of rewriting is the outer-most rewriting step for the object-programs which are object-terms with the `startfrom` type specified in the SL specification.

A failure exception is defined in the runtime libraries. The generated rewriting engines use the failure exception for matching. The code generated from a failure state raises such an exception.

### 2. Rewriting iteration function

The rewriting iteration function controls whether the evaluation should stop. The condition for terminating evaluation is that either the term is already in normal form or the number of iterations exceeds the given limit. There is a counter to record the number of iterations. Each time a reduction occurs, the counter increases by one.

More specifically, the rewriting iteration function is the entry point for reduction in all object-languages. The function takes the following argument: the number of iterations, the initial function for reduction, a printing function for object-terms, a pattern matching function, and a term to be rewritten.

### 3. Non-deterministic choosing function

The non-deterministic choosing function takes three arguments. The first is a list of alternatives. The second is the success continuation. The third is the choosing method. The choosing method can be one of the following:

- List order

The function chooses the alternatives one by one in the order appearing in the list. That is, the first alternative is chosen first. If the first alternative fails in matching, the next will be tried, and so on.

- Reverse order

The function chooses the alternatives in the reverse order in the list. In other words, the last alternative is chosen first. If the last alternative fails in matching, the next-to-last alternative will be tried, and so on.

- Random

The function randomly picks one alternative. If it fails in further matching, the function will make a random choice among the rest of the alternatives, and so on.

### Other utilities

Runtime libraries also provide utility functions that extend from standard CAML libraries. These include list utilities, name utilities, debug utilities, and printing utilities.

#### 1. List utilities

List utilities contain functions to handle lists, such as inserting one element at a particular place, extracting a specified element from a list, and rotating a matrix.

The functions provide extensions to the standard libraries in the CAML.

## 2. Name utilities

Name utilities contain functions to generate new names. The new names are composed by adding a suffix to a specified base. The suffix is an integer that represents the value of a counter. The counter is initialized to 0 and there are functions to set or reset the counter value. Each function call for newname increases the counter by one. For example, calling function newnamebase with “tmp” will produce the name “tmp\_0”. Subsequent calls will produce “tmp\_1”, and so on.

## 3. Debugging utilities

Debugging utilities contain functions that can optionally show the status of reduction or the values of important variables. There are also functions to control the level of verbosity during the reduction.

## 4. Printing utilities

Printing utilities contain functions to pretty print runtime terms, and to print reduction traces.

## 5. Main function

All the interpreters share the same main function code. The main function performs the command line options, calls the top function to parse the input terms, and invokes the rewrite engine to reduce the terms.

### Example of code generated by SL

We now describe the code generated from the specification for the call-by-value  $\lambda$ -calculus. The generated codes contain the following parts:

#### 1. Datatype definition and conversion.

Datatype definitions in the SIGNATURE part for the object-language are replicated into the generated code. For each datatype declared in the SIGNATURE part, functions to convert between terms with the type and runtime terms are generated.

In our example, the only datatype in the SIGNATURE part is  $M$ . Hence,  $M$  is declared as a datatype and functions `rtexpr2M` and `M2rtexpr` are conversion functions that implement the transformations show in Figure 7. Also, the example has a wrapper function that prints object-terms with type  $M$ .

```

type M = Var of string | Abs of (string* M) | App of (M* M)
;;

let rec rtexpr2M tname rte = ...
and M2rtexpr e = ...
;;

let sp_print_term sp_t_14 =
  (print_rtexpression (M2rtexpr sp_t_14))
;;

```

## 2. User functions

This part includes the CAML phrases given in the SPECIFICATION part. In our example, the subst function is replicated from the SL specification to the generated code.

```
(* The code in function part of source program *)

#open "namesupply";

let rec subst (t1, x, t2) = ...
;;
```

## 3. Automata for dynamic values or contexts

For each declared dynamic value or context, the definition is replicated into the generated code as comments for easy reference and comparison. There are matching functions generated from the automaton corresponding to the dynamic value or the context. Each state in the automaton contributes a function in the generated code. The function takes two arguments. One is the term to match and the other one is the success continuation. The entry point for the matching function is `sp_matchNAME_0`, which corresponds to the start state of the automaton. The NAME is the dynamic name or the context name.

In our example, `sp_matchValue_0` is the matching function for dynamic *Value*. We can see that in function `sp_matchValue_0`, it tries to use the matching construct in CAML to check whether the term has a constructor `Abs` at the top. If so, it calls the function `sp_matchValue_1` which corresponds to the accept state. Otherwise, it raises the matching failure exception.





```

and sp_matchE_1 sp_t_0 sk =
  ...

and sp_matchE_12 sp_t_0 sk =
  (let sp_t_0 = sp_t_0 in
   (sk
    ((function sp_t_0 ->
      (let sp_t_0 = sp_t_0 in sp_t_0)), sp_t_0)))

```

#### 4. Code for rewriting rules

Code for the rewriting rules is organized with the rule groups that are collected in compiling the SL specification. The code contains initialization functions and matching functions for all groups.

The initialization functions reset the list of used rules and reset the term tables for each group. The list of used rules is for the purpose of showing which rules are used in a deduction. Each rule group has a term table. The table records all the terms that already matched with the rule group.

Most of the functions generated in this part are matching functions for rule groups. Like the matching functions for dynamic values and contexts, these functions correspond to the states in the automata for the rule groups. Each function takes two arguments. One is the term to match and the other one is the success continuation. The entry point for the matching function for each rule group is `sp_matchNAME_0`, which corresponds to the start state of the automaton. The `NAME` is the rule group name.

In our example, we have two rule groups. The rule group `rg_0` contains the inference rule and the rule group `rg_1` contains both the inference rule and the axiom.

```

let rgtbl_sp_rg_0 = ref ([] : (T) list);;
let rgtbl_sp_rg_1 = ref ([] : (T) list);;

let init_1step () =
  used_rules := [];
  rgtbl_sp_rg_0 := [];
  rgtbl_sp_rg_1 := [];
;;

(* The code for rulegroup sp_rg_0 *)
let rec
sp_matchsp_rg_0_0 sp_t_1 sk =
(sp_sequence (sp_matchsp_rg_0_3 sp_t_1) (sp_matchsp_rg_0_1 ) sk)

and sp_matchsp_rg_0_3 sp_t_1 sk = (sp_matchE_0 sp_t_1 sk)

and sp_matchsp_rg_0_1 sp_t_1 sk =
  (let (sp_t_2, sp_t_3) = sp_t_1 in
   (sp_matchsp_rg_0_2 sp_t_2 sp_t_3 sk))

and sp_matchsp_rg_0_2 sp_t_2 sp_t_3 sk =
  (let e = sp_t_2 in

```

```

(let t1 = sp_t_3 in
  (let sp_t_1 = (sp_t_2 sp_t_3) in
    (sk
      (let _ = (add_usedrule "eval") in
        (try begin
          match (rewrite_1step sp_matchsp_rg_1 t1) with
            (t2, sp_t_0) ->
              ((e t2), (make_infer_pftree 1 "eval" [] [sp_t_0]))
          end
        with SP_Fail ->
          ((remove_usedrule()); (raise SP_Fail)))))))))

and sp_matchsp_rg_0 term =
  if mem term !rgtbl_sp_rg_0
  then raise SP_Fail
  else
    begin
      rgtbl_sp_rg_0 := term::!rgtbl_sp_rg_0;
      sp_matchsp_rg_0_0 term
    end

and (* The code for rulegroup sp_rg_1 *)

sp_matchsp_rg_1_0 sp_t_3 sk = ...

and sp_matchsp_rg_1_11 sp_t_3 sk = (sp_matchE_0 sp_t_3 sk)

```

```

and sp_matchsp_rg_1 term =
  if mem term !rgtbl_sp_rg_1
  then raise SP_Fail
  else
    begin
      rgtbl_sp_rg_1 := term::!rgtbl_sp_rg_1;
      sp_matchsp_rg_1_0 term
    end
  ;;

```

## 5. Miscellaneous

This part of the code includes functions invoking rewrites and calling main functions.

The function `start_rewrite` is the function that starts reduction on the given term. It calls the rewrite utility function and provides an initialization function, printing function, and matching function that are specific to the object-language. The reduction sequence is printed out by the runtime utilities if the trace flag is set. If the trace flag is not set, the `start_rewrite` prints the final result of reduction.

At the end of the code, there is a call to the main function defined in the runtime libraries.

```

let start_rewrite i term =
  sp_print_term term;
  print_newline();

```

```

let term1 =
  rewrite i init_1step (sp_print_term) sp_matchsp_rg_0 term;
in
print_newline();
if not (get_traceflag())
  then
  begin
  print_string " ==>> ";
  print_newline();
  sp_print_term term1;
  print_newline();
  end;
term1
;;

printexc__f
  main_control (init_func, compile_term_phrase); exit 0
;;

```

### Summary and Discussion

The SL system uses the first-order types of functional languages for specifying abstract syntax. We have not followed the approach of *higher-order abstract syntax (HOAS)* [PE88]. The HOAS representation would allow the variables of the object-language to be represented as meta-variables, and hence alleviates the need for explicitly reasoning about variable renaming and substitution of variables. However, higher-order abstract syntax interacts poorly with the inductive reasoning techniques that are needed

to reason about the properties of semantic specifications [DPS97].

The SL system uses conditional rewriting rules for specifying semantic rules. In other words, the system employs rewriting semantics (*a.k.a.* reduction semantics). Some systems[HM89, BCD<sup>+</sup>88] express rules in natural semantics[BH91, Kah87]. Natural semantics is well adapted to describing static behaviors such as typing. For transformational behaviors, such as dynamic semantics, rewriting semantics proves to be more modular[WF94], and should therefore be more tractable when it comes to expressing full-size programming languages. Another advantage of rewriting semantics is that it allows one to observe intermediate states of reductions, so that it is more suitable for non-terminating object-systems.

The ELAN system[ELA] is also based on first-order rewriting semantics. It has more general application areas than the SL system. It supports many-to-one associative-commutative(AC) patterns and provides an efficient algorithm[MK98]. Its strategy language gives flexible control over non-deterministic reductions. The Stratego[str, Vis01] system has more generic strategy specifications. There are also general-purpose tools that can be used for manipulating formal semantics, such as Coq[Coq], Isabelle[Isa] and Twelf[Twe]. When compared to those systems, the novelty of the SL system is that it directly supports the specification of semantic notions such as dynamic values and evaluation contexts, and it automatically generates executable interpreters. Associative-commutative patterns can be represented by contexts, but the current SL system does not optimize matching for efficiency.

The SL system described in this Chapter constitutes a first step towards designing a specification language for syntactic theories and implementing a system generating interpreters from specifications. It also provides an extension to pattern-matching tech-

niques. A number of interesting examples have been tested in the prototype system.



## CHAPTER III

### AUTOMATICALLY PROVING DECOMPOSITION

In almost every semantic treatment of programming languages that is based on manipulating syntax, one is immediately required to prove a *decomposition lemma* which shows the equivalence of two alternative syntactic specifications of the language: one tailored for exposing the syntactic structure of the language, and the other reflecting the relevant semantic notions, *e.g.*, values, redexes, head normal forms, untypable programs, and evaluation contexts.

In every natural case, the proof of the decomposition lemma is by induction on the structure of  $\Lambda$ -terms as given in Chapter I. Such a proof is often straightforward: it would typically be omitted from books or published articles. However, as discovered in several current projects [AF97, FLS99, LS97, SS99], such a proof is incredibly tedious in all but the most trivial programming languages. Moreover, in the process of designing the semantics one typically experiments with several definitions for the semantic notions. Every such experiment requires a laborious and error-prone proof attempt. A failed proof attempt typically reveals a missing case in one of the definitions. Even when one settles on the correct semantic notions, a successful proof might require several iterations where in each iteration one reformulates the syntactic definitions to strengthen the inductive hypothesis.

A further complication arises when the decomposition of a term must be *unique*. This is needed, for example, to derive a deterministic evaluation function, *i.e.*, an interpreter. The proof of a unique-decomposition lemma is usually performed by using

routine induction as before to find *some* decompositions, and then arguing that no further decompositions are possible. Published articles usually omit the complete proof of uniqueness as well. And again, arguing that no further decompositions are possible makes an already long and tedious proof even more intractable.

The obvious solution we seek is to automate the proof of unique decomposition. Given the two syntactic definitions of interest, we would like an algorithm that either confirms the equivalence of the two definitions or produces a counterexample. A counterexample is a term that has no decomposition, or in the uniqueness case, a term with more than one decomposition. Proving a decomposition lemma reduces to checking the equivalence of two grammars and proving uniqueness reduces to checking that a grammar is not ambiguous. However, both equivalence and ambiguity are undecidable in general. We thus restrict the syntactic definitions to regular tree grammars, for which the problems are decidable. Unfortunately, regular tree grammars are not quite expressive enough to accommodate contexts, polymorphic types, and other constructs usually needed in definitions of syntactic theories. We therefore engineer an extension of the framework that is expressive enough while retaining the decidability of the fundamental algorithms.

A prototype of the above checking is implemented in the SL system and has been tested on a number of examples.

The remainder of this chapter is organized as follows: We first give a manual proof of unique-decomposition for the call-by-value  $\lambda$ -calculus. Then we show the relationship between proofs of unique-decomposition lemmas and the problems of equivalence and ambiguity of grammars. After presenting a survey of the needed concepts about regular tree grammars and finite tree automata. We adapt those concepts to the develop-

ment of algorithms for proving unique-decomposition lemmas. We describe extensions to this approach by considering built-in constants, tuples, contexts, and polymorphic types, and explaining how to construct tree automata for them. Next we introduce an extension to the SL system for automatically proving unique-decomposition lemmas. We then summarize and review related work.

### Manual Proof of Unique Decomposition

#### Lemma 3 (Unique Decomposition)

Every term  $M$  can be uniquely decomposed into one of the following forms: a value  $V$ , or a demand for a variable  $x$  within an evaluation context  $H$ , or a redex  $R$  within an evaluation context  $E$ .

**Proof:** We first prove that every term has *some* decomposition. The proof is by induction on  $M$  and proceeds by case analysis:

- **Case  $M = x$ :** Then  $M$  is of the form  $E[x]$  where  $E = []$ .
- **Case  $M = \lambda x.M_1$ :** Then  $M$  is of the form  $V$ .
- **Case  $M = M_1M_2$ :** By induction, the expression  $M_1$  can be decomposed in one of the following four forms:
  - **Case  $M_1 = V$ :** There have two subcases:
    - **Case  $M_1 = \lambda x.M_3$ :** Then we proceed by induction on  $M_2$ :
      - **Case  $M_2 = V$ :** Then  $M = (\lambda x.M_3)V$ , which is of the form  $E[R]$  with  $E = []$ .

- **Case  $M_2 = E[x]$ :** Then  $M = (\lambda x.M_3)E[x]$ , which is of the form  $E'[x]$  with  $E' = VE$ .
- **Case  $M_2 = E[R]$ :** Then  $M = (\lambda x.M_3)E[R]$ , which is of the form  $E'[R]$  with  $E' = VE$ .
- **Case  $M_1 = E[x]$ :** Then  $M = E[x]M_2$  which is of the form  $E'[x]$  with  $E' = EM_2$ .
- **Case  $M_1 = E[R]$ :** Then  $M = E[R]M_2$  which is of the form  $E'[R]$  with  $E' = EM_2$ .

To prove that every term has a unique decomposition, we proceed by first showing that some decomposition is possible as before and then once a decomposition is found, arguing that no further decompositions exist.  $\square$

### Equivalence and Ambiguity of Grammars

Abstractly speaking, the problem of unique decomposition consists of relating two syntactic definitions for the same language. Generally, these syntactic presentations would be given as context-free grammars  $G_1$  and  $G_2$ . Unique decomposition [Bar84, Corollary 8.3.8] can then be restated as problems about grammars  $G_1$  and  $G_2$ :

- The existence of a decomposition can be reduced to checking that every term derived by  $G_1$  can be derived by  $G_2$  as well. Since the other implication is usually trivial, the existence of a decomposition is reduced to checking the equivalence of grammars  $G_1$  and  $G_2$ , which is undecidable [HU79, Theorem 8.12].
- The uniqueness of a decomposition can be reduced to checking that every term in  $G_1$  can only be derived by  $G_2$  in a unique way. In other words, uniqueness requires

checking whether grammar  $G_2$  is unambiguous, which is also undecidable [HU79, Theorem 8.9].

Therefore, we cannot hope to solve the problem of unique decomposition automatically for syntactic definitions given as context-free grammars. Fortunately, syntactic definitions of programming languages are usually given using constructors, and the constructors impose structural restrictions on terms. These syntactic definitions can fit in a class of grammars, called *regular tree grammars*, for which checking equivalence and checking ambiguity are decidable.

### Regular Tree Grammars and Finite Tree Automata

We review the necessary background on regular tree grammars and their computational models in this section.

#### Regular Tree Grammars

A regular tree grammar [GS84] is a context-free grammar with one important restriction: the right-hand sides of the productions can only be generated using a fixed number of constructors. The formal definitions are as follows:

##### Definition 1

Let  $\Sigma$  be a finite set of constructors and let  $\Phi$  be a finite set of variables. The set of *trees* over  $\Sigma$  and  $\Phi$ , denoted as  $\mathcal{T}(\Sigma, \Phi)$ , is inductively defined as follows:

- $\alpha$  is a tree, where  $\alpha \in \Phi$ .

- If  $t_1, \dots, t_n$  are trees, then  $c^n(t_1, \dots, t_n)$  is a tree, where  $n \geq 0$  and  $c^n$  is a constructor of arity  $n$  in  $\Sigma$ .

We sometimes blur the distinction between *terms* and *trees* in this chapter, although terms are typically defined over infinite sets of constants and variables. Terms without variables are called *ground terms*.

### Definition 2

A *regular tree grammar* is a quadruple of the form  $(\Sigma, N, S, P)$  where:

- $\Sigma$  is a finite set of constructors.
- $N$  is a finite set of non-terminals.
- $S$  is the start symbol,  $S \in N$ .
- $P$  is a finite set of rules (productions) of the form  $L \rightarrow s$  where  $L$  is a non-terminal and  $s \in \mathcal{T}(\Sigma, N)$ .

If the same non-terminal appears as the left-hand side of several productions, we merge all the productions into one clause using alternatives. Non-terminals do not all necessarily derive ground terms. In the remainder of this chapter, we assume that all non-terminals can be derived from the start symbol.

### Example 1

A regular tree grammar for  $\lambda$ -calculus is:

- $\Sigma$  is the set  $\{\text{var}, \text{lam}, \text{app}\}$ .
- $N$  is the set  $\{M\}$ .
- $S$  is  $M$ .

- $P$  is the set:

$$M \rightarrow \text{var} \mid \text{lam}(\text{var}, M) \mid \text{app}(M, M)$$

The grammar has been modified from the original presentation in Chapter I by using terminal symbols such as `lam` and `var`. The infinite set of variables is abstracted into `var`. This abstraction is usually possible since no reference to individual variables is made in most semantic specifications.<sup>1</sup>

### Example 2

A regular tree grammar expressing the decomposition of  $\lambda$ -calculus terms into head normal forms and terms with a head redex is:

- $\Sigma$  is the set  $\{\text{var}, \text{lam}, \text{app}\}$ .
- $N$  is the set  $\{M, H, I, U, R\}$ .
- $S$  is  $M$ .
- $P$  is the set:

$$M \rightarrow H \mid U$$

$$H \rightarrow I \mid \text{lam}(\text{var}, H)$$

$$I \rightarrow \text{var} \mid \text{app}(I, M)$$

$$U \rightarrow R \mid \text{lam}(\text{var}, U)$$

$$R \rightarrow \text{app}(\text{lam}(\text{var}, M), M) \mid \text{app}(R, M)$$

---

<sup>1</sup>The specification for the call-by-need calculus is an exception [AF97].

Some non-terminals are introduced in the regular tree grammar for decompositions to avoid using the informal abbreviation "...".

### Finite Tree Automata

The algorithms for checking equivalence and ambiguity of regular tree grammars are described using the notion of *finite tree automata* defined as follows:

#### Definition 3 (FTA)

A *finite tree automaton* is a quadruple  $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$  where

- $\Sigma$  is a finite set of constructors.
- $Q$  is a finite set of states, ranged over by  $q, q_1, q_2, \dots$
- $Q_f \subseteq Q$  is a set of final states.
- $\Delta$  is a finite set of transition rules, each in one of the following forms:
  - \*  $q_1 \xrightarrow{\epsilon} q_2,$
  - \*  $c^n(q_1, \dots, q_n) \rightarrow q$  where  $n \geq 0$  and  $c^n$  is a constructor of arity  $n$  in  $\Sigma$ .

An automaton can be represented by its transition rules and final states when other information can be easily collected from the transition rules. If the final states are obvious to determine, they can be omitted as well.

#### Example 3

A possible tree automaton corresponding to the regular tree grammar of



Example 1 is:

$$\begin{array}{lcl}
 q_{M_1} \xrightarrow{\epsilon} q_M & & \text{var} \rightarrow q_{M_1} \\
 q_{M_2} \xrightarrow{\epsilon} q_M & \text{lam}(q_{M_4}, q_M) \rightarrow q_{M_2} & \text{var} \rightarrow q_{M_4} \\
 q_{M_3} \xrightarrow{\epsilon} q_M & \text{app}(q_M, q_M) \rightarrow q_{M_3} & 
 \end{array}$$

where  $q_{M_1}, q_{M_2}, q_{M_3}$  correspond to the three alternatives of  $M$ .

#### Definition 4

Given a finite tree automaton  $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$ , the  $\epsilon$ -transition closure of states  $\{q_1, \dots, q_n\}$ , denoted by  $cls_\epsilon\{q_1, \dots, q_n\}$ , is the set of states  $\{q \mid q_i \xrightarrow{\epsilon^*} q, 1 \leq i \leq n\}$ , where  $\xrightarrow{\epsilon^*}$  is the reflexive, transitive closure of  $\xrightarrow{\epsilon}$ .

#### Definition 5

Given an automaton  $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$ ,

- a run<sup>2</sup>  $r$  of  $\mathcal{A}$  on a ground term  $t$  is a total mapping from subterms of  $t$  to  $\epsilon$ -transition closures of the singletons  $\{q\}$ , such that  $r$  is compatible with  $\Delta$ , i.e., for every subterm  $p$ , if  $p = c^n(p_1, \dots, p_n)$ , where  $c^n \in \Sigma$ ,  $n \geq 0$ ,  $r(p) = cls_\epsilon\{q\}$  and  $r(p_i) = cls_\epsilon\{q_i\}$ , for each  $1 \leq i \leq n$ , then there exist  $q' \in cls_\epsilon\{q\}$  and  $q'_i \in cls_\epsilon\{q_i\}$ , for each  $1 \leq i \leq n$ , such that  $c^n(q'_1, \dots, q'_n) \rightarrow q' \in \Delta$ .
- a run  $r$  of  $\mathcal{A}$  on term  $t$  is *successful* if  $r(t)$  contains a final state.

---

<sup>2</sup>Our definition of run is more general than the standard definition which is based on automata without  $\epsilon$ -transitions. We keep  $\epsilon$ -transitions because they will be used in proving uniqueness.

- a term  $t$  is *recognized* (or *accepted*) by an automata  $\mathcal{A}$  if there exists a successful run of  $\mathcal{A}$  on  $t$ .
- a term  $t$  can *reach* a state  $q$  if there exists a run  $r$  of  $\mathcal{A}$  on  $t$  such that  $q \in r(t)$ .

Intuitively, a successful run on a ground term  $t$  tells us how  $t$  is constructed, *i.e.*, it corresponds to  $t$ 's parse tree. For example, a successful run  $r$  of the automaton given in Example 3 on  $\text{app}(\text{lam}(\text{var}_1, \text{var}_2), \text{var}_3)$ <sup>3</sup> could be defined as follows:

$$\begin{aligned}
 r(\text{var}_1) &= \{q_{M_1}\} \\
 r(\text{var}_2) &= \{q_{M_1}, q_M\} \\
 r(\text{lam}(\text{var}_1, \text{var}_2)) &= \{q_{M_2}, q_M\} \\
 r(\text{var}_3) &= \{q_{M_1}, q_M\} \\
 r(\text{app}(\text{lam}(\text{var}_1, \text{var}_2), \text{var}_3)) &= \{q_{M_3}, q_M\}
 \end{aligned}$$

As an example of  $r$ 's compatibility with the productions of  $\Delta$ , note that the production for the constructor  $\text{lam}$  maps the states  $q_{M_1}$  and  $q_M$  to  $q_{M_2}$ , and that these states are included in the images of  $\text{var}_1$ ,  $\text{var}_2$ , and  $\text{lam}(\text{var}_1, \text{var}_2)$ , respectively.

### Definition 6

The *tree language* recognized by an automaton  $\mathcal{A}$ , denoted by  $L(\mathcal{A})$ , is the set of all terms accepted by  $\mathcal{A}$ .

Two finite tree automata are *equivalent* if they recognize the same tree language. Like finite automata, finite tree automata are divided into deterministic finite tree automata (DFTAs) and non-deterministic ones (NFTAs). A finite tree automaton is *deter-*

---

<sup>3</sup>We use  $\text{var}_i$ ,  $i = 1, 2, 3$  to distinguish the different positions of  $\text{var}$ .

*ministic* if no two transition rules have the same left-hand side and there is no  $\epsilon$ -transition rule. Otherwise, it is *non-deterministic*. For instance, the automaton for  $\lambda$ -terms in Example 3 is non-deterministic.

Many theorems in finite automata theory have counterparts in finite tree automata theory. Here are some theorems [CDG<sup>+</sup>99] we will use later.

### Theorem 1

Given an NFTA, there exists an equivalent DFTA.

The algorithm for constructing a DFTA from an NFTA is similar to the classical subset construction algorithm for finite state automata [ASU85]. It groups together the non-deterministic states that have common terms reaching them and treats the groups as deterministic states. The final deterministic states are the groups containing at least one non-deterministic final state.

The union, intersection, complement, and difference of tree languages are still tree languages. There is also a corresponding pumping lemma for tree languages, a useful corollary of which is that testing emptiness of a tree language is decidable.

### Theorem 2

The class of tree languages is closed under union, intersection, complement, and difference.

### Theorem 3

Let  $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$  be a DFTA, then  $L(\mathcal{A})$  is not empty if and only if there exists a term  $t$  in  $L(\mathcal{A})$  with  $Height(t) \leq |Q|$ , where  $Height(t)$  is the height of  $t$ 's tree representation and  $|Q|$  is the number of states in  $Q$ .

### Algorithms for Proving Unique Decomposition

Our approach for automatically proving unique-decomposition lemmas consists of translating the regular tree grammars for both terms and decompositions into finite tree automata, and checking the existence and the uniqueness of decompositions using corresponding automata algorithms.

#### Building Finite Tree Automata

Each regular tree grammar can be mapped to a finite tree automaton using the following translation. Our algorithms for checking decompositions are specialized to the output of this translation.

- If there are alternative production rules for the same non-terminal, introduce a non-terminal for each alternative, and add a production rule from the original non-terminal to the newly introduced ones. The reason alternative productions are distinguished in this way is to simplify the check for non-overlapping productions required for proving uniqueness. For example, the production rules of  $M$  in Example 1 are turned into:

$$\begin{array}{l} M \rightarrow M_1 \quad M_1 \rightarrow \text{var} \\ M \rightarrow M_2 \quad M_2 \rightarrow \text{lam}(\text{var}, M) \\ M \rightarrow M_3 \quad M_3 \rightarrow \text{app}(M, M) \end{array}$$

- Take apart the right-hand sides of the production rules in such a way that each right-hand side is a non-terminal, a constructor of arity zero, or an application of a constructor to non-terminals. For example, the production rule  $M_2 \rightarrow \text{lam}(\text{var}, M)$

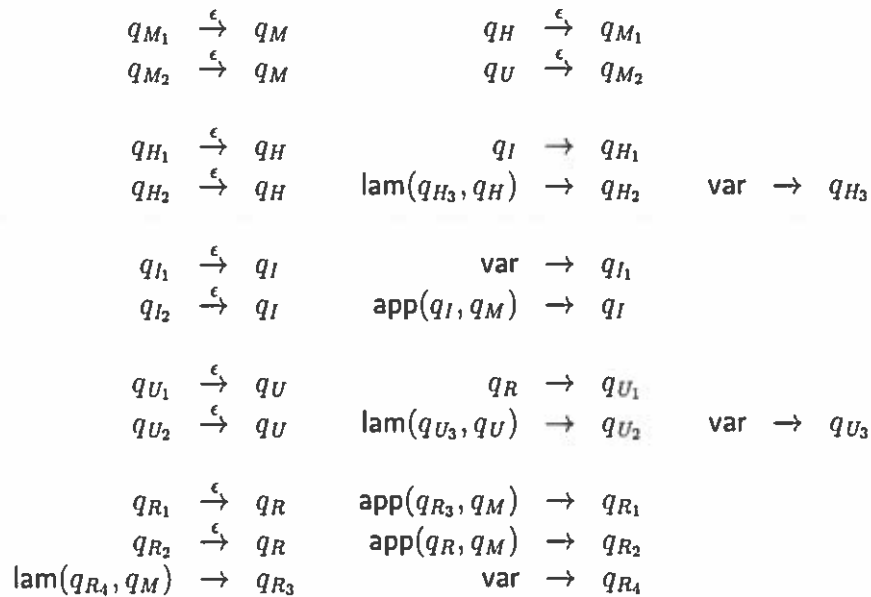


FIGURE 8. An automaton for the grammar of Example 2

is turned into  $M_2 \rightarrow \text{lam}(M_4, M)$  and  $M_4 \rightarrow \text{var}$ .

- Reverse the arrows of the production rules and introduce a state for each non-terminal. The state corresponding to the start symbol is the final state.

Example 3 shows the automaton translated from the regular tree grammar for  $\lambda$ -calculus from Example 1. The automaton for decompositions built from the tree grammar in Example 2 is given in Figure 8, where  $q_M$  is the final state.

The automaton produced by the above translation is usually non-deterministic since most grammars have alternatives and the alternatives create  $\epsilon$ -transitions. This non-deterministic automaton is then converted to a deterministic automaton. It is during this translation that the checking of uniqueness of decomposition is made. The existence of a decomposition is then established using the deterministic automaton.

### Checking Existence of Decompositions

Given two deterministic automata:  $\mathcal{A}_1$  for the terms and  $\mathcal{A}_2$  for the decompositions, we can answer whether all terms have decompositions by checking whether  $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$ , which is equivalent to checking whether  $L(\mathcal{A}_1) - L(\mathcal{A}_2)$  is empty. By Theorem 2, there exists an automaton  $\mathcal{A}_1 - \mathcal{A}_2$  whose language is  $L(\mathcal{A}_1) - L(\mathcal{A}_2)$ . In order to check the emptiness of  $L(\mathcal{A}_1) - L(\mathcal{A}_2)$ , according to Theorem 3, it is sufficient to check whether the automaton  $\mathcal{A}_1 - \mathcal{A}_2$  accepts a term with height less than or equal to its number of states. We thus generate all such terms and check whether they are accepted. If none is accepted, then  $L(\mathcal{A}_1 - \mathcal{A}_2)$  is empty, *i.e.*,  $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$ , and all terms have decompositions. Otherwise, if a term is accepted by the automaton, then it is a counterexample without decomposition.

The number of terms being tested is  $O(|\Sigma|^{mk})$ , where  $|\Sigma|$  is the number of constructors,  $m$  is the maximal arity of the constructors in  $\Sigma$ , and  $k$  is the number of states in the automaton  $\mathcal{A}_1 - \mathcal{A}_2$ . The upper bound on  $k$  is obtained as follows:  $\mathcal{A}_1 - \mathcal{A}_2$  is equivalent to  $\mathcal{A}_1 \cap \overline{\mathcal{A}_2}$ , where  $\overline{\mathcal{A}_2}$  is the complement of  $\mathcal{A}_2$ . Suppose  $\mathcal{A}_1$  has  $k_1$  states and  $\mathcal{A}_2$  has  $k_2$  states, then by standard algorithms,  $\overline{\mathcal{A}_2}$  has  $k_2 + 1$  states and  $\mathcal{A}_1 \cap \overline{\mathcal{A}_2}$  has at most  $k_1 * (k_2 + 1)$  states. Usually  $k_1$  is greater than  $|\Sigma|$  and  $k_2$  may be much greater. Therefore  $k$  is greater than  $|\Sigma|^2$  in most cases and the algorithm is exponential in the number of constructors.

### Checking Uniqueness of Decompositions

Proving uniqueness corresponds to checking the non-ambiguity of the grammar for the decomposition patterns. If the grammar is ambiguous, there exists a term  $t$  with more than one parse tree which can only happen due to the presence of alternative pro-

ductions for the same non-terminal. Suppose that, in the non-deterministic automaton translated from the grammar, states  $q_1$  and  $q_2$  correspond to those two alternatives. This means that there are two successful runs of the non-deterministic automaton on  $t$  that map a subterm of  $t$  to two different  $\epsilon$ -transition closures containing  $q_1$  and  $q_2$ , respectively. In other words, the sets of terms reaching states  $q_1$  and  $q_2$  have a non-empty intersection. Such states, reached by overlapping sets of terms, can be found during the translation of the non-deterministic automaton into a deterministic one, as the translation naturally groups non-deterministic states by the common terms that can reach them.

To formalize the above algorithm, we first define conflict sets.

Definition 7

Given an automaton  $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$ , a *conflict set* of  $\mathcal{A}$  is a set  $C$  of at least two states, and a special state  $q \in Q$  such that:

$$q_c \in C \text{ if and only if } q_c \xrightarrow{\epsilon} q \in \Delta$$

DFTAs do not have conflict sets since they do not have  $\epsilon$ -transition rules. For general non-deterministic automata, conflict sets may exist for many reasons. But for the non-deterministic automata produced by our translation from regular tree grammars (as described in a previous section), conflict sets correspond to alternatives for non-terminals. For example, the automaton for lambda-calculus given in Example 3 has only one conflict set  $\{q_{M_1}, q_{M_2}, q_{M_3}\}$ , while the automaton for the decompositions in Figure 8 has conflict sets  $\{q_{M_1}, q_{M_2}\}$ ,  $\{q_{H_1}, q_{H_2}\}$ ,  $\{q_{I_1}, q_{I_2}\}$ ,  $\{q_{U_1}, q_{U_2}\}$ , and  $\{q_{R_1}, q_{R_2}\}$ .

Because of the special constraints on automata imposed by our construction, we can check uniqueness of decompositions by keeping track of conflict sets while translat-

ing NFTAs to DFTAs. Our algorithm checks whether a deterministic state contains more than one non-deterministic state from the same conflict set. If there does not exist such a deterministic state, then the grammar is not ambiguous and hence, the decomposition for any term (if one exists) is unique. Otherwise, counterexamples violating uniqueness can be obtained by the following steps: Let  $\mathcal{A}_N = (\Sigma, Q_N, Q_{Nf}, \Delta_N)$  be an NFTA and  $\mathcal{A}_D = (\Sigma, Q_D, Q_{Df}, \Delta_D)$  be the DFTA translated from  $\mathcal{A}_N$ .

1. For each deterministic state, construct a term that reaches it. Such a term is also called an *instance term* for the state. The instance terms are constructed bottom-up:
  - (a) If a deterministic state  $g$  has no instance term and if there exists  $c^0 \rightarrow g \in \Delta_D$ , then  $c^0$  is an instance term of  $g$ .
  - (b) If a deterministic state  $g$  has no instance term, states  $g_1, \dots, g_n$  have instance terms  $t_1, \dots, t_n$  respectively, and if there exists  $c^n(g_1, \dots, g_n) \rightarrow g \in \Delta_D$ , then  $c^n(t_1, \dots, t_n)$  is an instance term of  $g$ .
  - (c) Repeat (a) and (b) above until all deterministic states have one instance term each.
2. Suppose a deterministic state  $g$  contains  $q_1$  and  $q_2$ , which are from the same conflict set, and  $g$  has an instance term  $t$ . Let  $q$  be the state associated with the conflict set, i.e.,  $q_1 \xrightarrow{\epsilon} q \in \Delta_N$  and  $q_2 \xrightarrow{\epsilon} q \in \Delta_N$ . Initiate a queue  $L$  with an element  $(q, t)$ . Repeat the following steps until a counterexample is found:
  - (a) Get the first element out from  $L$ , let it be  $(q', t')$ .
  - (b) If  $q' \in Q_{Nf}$ , then the term  $t'$  is the counterexample.



(c) Otherwise:

- For each transition rule  $c^n(p_1, \dots, p_n) \rightarrow q'' \in \Delta_N$  where  $q' = p_i$ , let  $g_j$  be the deterministic state containing  $p_j$  for each  $j \neq i$ . Construct a term  $t'' = c^n(t_1, \dots, t_n)$  where  $t_j$  is the instance term of  $g_j$  for  $j \neq i$  and  $t_i = t'$ . Append  $(q'', t'')$  to the queue  $L$ .
- For each transition rule  $q' \xrightarrow{\epsilon} q'' \in \Delta_N$ , append  $(q'', t')$  into the queue  $L$ .

In the second step, we construct terms containing  $t$  as a subterm in bottom-up fashion following the productions of the automaton  $\mathcal{A}_N$ . Runs of  $\mathcal{A}_N$  on all the constructed terms map a subterm  $t$  to  $cls_\epsilon\{q_1\}$  or  $cls_\epsilon\{q_2\}$ . The construction ends when we obtain an instance term for a deterministic final state. Termination is guaranteed since we assume that all non-terminals can be derived from the start symbol of the grammar and traversing of the productions of  $\mathcal{A}_N$  is in breadth-first order.

The grammars seen so far are non-ambiguous. We show in the following example how the algorithm works on an ambiguous grammar.

#### Example 4

Consider the following regular tree grammar:

$$S \rightarrow \text{not}(\text{false}) \mid \text{not}(A)$$

$$A \rightarrow \text{or}(\text{true}, B) \mid \text{or}(B, \text{true})$$

$$B \rightarrow \text{true} \mid \text{false}$$

A NFTA for the grammar is:

$$q_{S_1} \xrightarrow{\epsilon} q_S \quad \text{not}(q_{S_3}) \rightarrow q_{S_1} \quad \text{false} \rightarrow q_{S_3}$$

$$q_{S_2} \xrightarrow{\epsilon} q_S \quad \text{not}(q_A) \rightarrow q_{S_2}$$

$$q_{A_1} \xrightarrow{\epsilon} q_A \quad \text{or}(q_{A_3}, q_B) \rightarrow q_{A_1} \quad \text{true} \rightarrow q_{A_3}$$

$$q_{A_2} \xrightarrow{\epsilon} q_A \quad \text{or}(q_B, q_{A_4}) \rightarrow q_{A_2} \quad \text{true} \rightarrow q_{A_4}$$

$$q_{B_1} \xrightarrow{\epsilon} q_B \quad \text{true} \rightarrow q_{B_1}$$

$$q_{B_2} \xrightarrow{\epsilon} q_B \quad \text{false} \rightarrow q_{B_2}$$

where the conflict sets are  $\{q_{S_1}, q_{S_2}\}$ ,  $\{q_{A_1}, q_{A_2}\}$ , and  $\{q_{B_1}, q_{B_2}\}$ .

The above NFTA is translated into the following DFTA:

$$\text{true} \rightarrow g_1 = \{q_{A_3}, q_{A_4}, q_{B_1}, q_B\}$$

$$\text{false} \rightarrow g_2 = \{q_{S_3}, q_{B_2}, q_B\}$$

$$\text{or}(g_1, g_1) \rightarrow g_3 = \{q_{A_1}, q_{A_2}, q_A\}$$

$$\text{or}(g_1, g_2) \rightarrow g_4 = \{q_{A_1}, q_A\}$$

$$\text{or}(g_2, g_1) \rightarrow g_5 = \{q_{A_2}, q_A\}$$

$$\text{not}(g_2) \rightarrow g_6 = \{q_{S_1}, q_S\}$$

$$\text{not}(g_3) \rightarrow g_7 = \{q_{S_2}, q_S\}$$

$$\text{not}(g_4) \rightarrow g_7$$

$$\text{not}(g_5) \rightarrow g_7$$

The deterministic state  $g_3$  contains  $q_{A_1}$  and  $q_{A_2}$  which are from the same conflict set. The state  $g_3$  has an instance term  $\text{or}(\text{true}, \text{true})$ . We need

to construct a term that is accepted by the automaton and that contains the instance term. The state  $q_A$  is the special state associated with the conflict set  $\{q_{A_1}, q_{A_2}\}$ . By following the non-deterministic transition rule  $\text{not}(q_A) \rightarrow q_{S_2}$ , we obtain the term  $\text{not}(\text{or}(\text{true}, \text{true}))$  which can reach  $q_{S_2}$ . Then by following the rule  $q_{S_2} \xrightarrow{\xi} q_S$ , the term can also reach the final state  $q_S$ . In other words, the term  $\text{not}(\text{or}(\text{true}, \text{true}))$  is a counterexample to uniqueness.

### Extensions

Regular tree grammars are expressive enough for specifying terms and decompositions built only from constructors. However, in specifying unique-decomposition lemmas, we often use contexts and polymorphic types. These semantic notions cannot be directly defined by regular tree grammars. In order to handle definitions that use contexts and polymorphic types, we extend the definition of tree automata with variables that stand for unknown subterms and identify necessary restrictions for maintaining decidability.

We first give simple extensions by considering built-in constants and tuples. Then we explain how to build automata for contexts and polymorphic types.

### Built-in Constants

Some built-in types such as `int` and `string` are infinite. As we saw previously, we can collapse an infinite type into a constant. For example, we collapse `int` into `int` and `string` into `str`. The transition rule  $\text{int} \rightarrow q$  means that all integer numbers can reach state  $q$ . It is a substitute for an infinite number of transition rules  $0 \rightarrow q, 1 \rightarrow q,$

$\dots, n \rightarrow q$ . In this way, we can use finite constructors and transition rules to recognize elements of infinite types. This method works for specifications without specific built-in constants such as the examples considered so far, but the following definition requires distinguishing 0 from other constants.

#### Example 5

The following grammar defines integer lists with at most one occurrence of 0 and where the occurrence can be only at the head.

$$\begin{aligned} G &\rightarrow \text{cons}(\text{int}, I) \\ I &\rightarrow \text{nil} \mid \text{cons}(n, I) \quad \text{where } n \neq 0 \end{aligned}$$

We divide the infinite set such that the constants appearing in the specification are kept distinct, while those not appearing in the specification are collapsed into one special constant. The special constants are represented as  $\text{allexcept}(T, \{c_1, \dots, c_n\})$ , where  $\text{allexcept}$  is a tag,  $T$  is an infinite built-in type, and  $c_1, \dots, c_n$  are constants in the type  $T$  appearing in the specification. In other words, we add a production rule

$$T \rightarrow c_1 \mid \dots \mid c_n \mid \text{allexcept}(T, \{c_1, \dots, c_n\})$$

in the grammars and move  $T$  from the set of terminals to the set of non-terminals. This change preserves the set of terms that can be derived from the grammar.

In this setting, an automaton for the grammar in Example 5 is:

$$\begin{aligned}
 \text{cons}(q_{int}, q_I) &\rightarrow q_G \\
 0 &\rightarrow q_{int} & \text{allexcept}(int, \{0\}) &\rightarrow q_{int} \\
 \text{nil} &\rightarrow q_I & \text{cons}(q_{I_1}, q_I) &\rightarrow q_I \\
 \text{allexcept}(int, \{0\}) &\rightarrow q_{I_1}
 \end{aligned}$$

### Tuples

Tuples are frequently used in syntactic definitions. To build automata for definitions with tuples, we introduce a new constructor tuple. The arity of tuple is not fixed. The constructor tuple stands for any constructor tuple<sup>2</sup>, tuple<sup>3</sup>, ..., tuple<sup>n</sup>, where tuple<sup>n</sup> has arity  $n$ ,  $n \geq 2$ .

#### Example 6

A pair of boolean values can be decomposed into four cases. The grammars are specified as follows:

$$\begin{aligned}
 G_1 &\rightarrow (bool, bool) \\
 G_2 &\rightarrow (true, true) \mid (true, false) \mid (false, true) \mid (false, false)
 \end{aligned}$$

The automaton for  $G_1$  has the following transition rules, where  $q_{G_1}$  is the final state:

$$\text{tuple}(q_{bool}, q_{bool}) \rightarrow q_{G_1} \quad \text{true} \rightarrow q_{bool} \quad \text{false} \rightarrow q_{bool}$$

## Contexts

Contexts are usually used to identify the location of a subterm in a term. For instance, a call-by-value computation consists of successively applying the  $\beta_v$ -reduction rule to any subterm. In order to define a deterministic evaluation function, *i.e.*, an interpreter, we restrict the application of the reduction rule to a unique subterm of a program whose position is determined using an evaluation context.

### Example 7

The set of call-by-value evaluation contexts has the following definition:

$$E ::= [] \mid E M \mid (\lambda x.M) E$$

where  $M$  stands for the set of  $\lambda$ -terms defined in Example 1.

The  $[]$  represents a “hole”. If  $E$  is a context, then  $E[t]$  denotes the term that results from placing  $t$  in the hole of  $E$ .

### Abstract Automata

Our approach is to represent the holes as (meta-)variables and to represent context as terms with (meta-)variables. The exposition in “Computing with Contexts” [Mas99] provides a more complete investigation of how to compute with contexts in general settings. The finite tree automata cannot handle terms with (meta-)variables. To address this issue, we introduce a new representation of a tree automaton which is a quadruple  $(\Sigma, (Q, \Phi_c), Q_f, \Delta)$ , where  $\Phi_c$  is a (finite) set of the states that do not appear in the right-hand side of any transition rule. The sets  $Q$  and  $\Phi_c$  are disjoint and their union is the set of all states in the automaton. We call the states in  $\Phi_c$  *state variables* and call

automata with a non-empty set of state variables *abstract automata*. All the theorems in the survey section still hold in the presence of  $\Phi_c$ . The definition of run will then map a variable occurring in a term to the  $\epsilon$ -transition closure of a state variable. Different variables should be mapped to different closures and different occurrences of the same variable should be mapped to the same closure.

### Example 8

An abstract automaton for the evaluation contexts  $E$  of Example 7 is:

$$\begin{array}{ll}
 q_{E_1} \xrightarrow{\epsilon} q_E & \alpha \xrightarrow{\epsilon} q_{E_1} \\
 q_{E_2} \xrightarrow{\epsilon} q_E & \text{app}(q_E, q_M) \rightarrow q_{E_2} \\
 q_{E_3} \xrightarrow{\epsilon} q_E & \text{app}(q_{E_4}, q_E) \rightarrow q_{E_3} \\
 \text{lam}(q_{E_5}, q_M) \rightarrow q_{E_4} & \text{var} \rightarrow q_{E_5}
 \end{array}$$

where  $\alpha$  is the only state variable and  $\{q_{E_1}, q_{E_2}, q_{E_3}\}$  is the conflict set. The transition rules for  $q_M$  are omitted.

A context  $(\lambda x.x) []$  can be represented as the term  $\text{app}(\text{lam}(\text{var}_1, \text{var}_2), \alpha)$ .

A successful run  $r$  of the automaton for this term is:

$$\begin{aligned}
 r(\text{var}_1) &= \{q_{E_5}\} \\
 r(\text{var}_2) &= \{q_{M_1}, q_M\} \\
 r(\text{lam}(\text{var}_1, \text{var}_2)) &= \{q_{E_4}\} \\
 r(\alpha) &= \{\alpha, q_{E_1}, q_E\} \\
 r(\text{app}(\text{lam}(\text{var}_1, \text{var}_2), \alpha)) &= \{q_{E_3}, q_E\}
 \end{aligned}$$

The operation  $E[t]$  that fills a context  $E$  with term  $t$  corresponds to instantiating the state variables of the automaton for  $E$  as follows: Instantiations add  $\epsilon$ -transitions from the final states in the automaton for the term  $t$  to the state variables which become normal states of the automaton for  $E$ . In order to avoid conflicts in state names, instantiations usually involve replications of the automata. Given an abstract automaton  $\mathcal{A} = (\Sigma, (Q, \Phi_c), Q_f, \Delta), \{\alpha_1, \dots, \alpha_n\} \subseteq \Phi_c$  and the automata  $\mathcal{A}_i = (\Sigma_i, (Q_i, \Phi_{ci}), Q_{fi}, \Delta_i)$  for each subterm  $t_i, 1 \leq i \leq n$ , instantiating  $\alpha_i$  with  $t_i$ , for all  $1 \leq i \leq n$ , involves the following steps:

1. Make a copy of  $\mathcal{A}$ , obtaining  $\mathcal{A}' = (\Sigma', (Q', \Phi'_c), Q'_f, \Delta')$ .
2. Rename states in  $\mathcal{A}_i$  for  $1 \leq i \leq n$  if necessary, so that there is no name conflict among states.
3. Combine  $\mathcal{A}'$  and  $\mathcal{A}_i$ , for  $1 \leq i \leq n$ , but keep  $Q'_f$  as the set of final states.
4. Add transition rules  $q_{fi} \xrightarrow{\epsilon} \alpha'_i$ , where  $q_{fi} \in Q_{fi}$ . Move  $\alpha'_1, \dots, \alpha'_n$  from the set of state variables to the set of normal states.

The resulting automaton has the following components:

- constructors:  $\Sigma' \cup \bigcup_{i=1}^n \Sigma_i$
- states:  $(Q' \cup \{\alpha'_1, \dots, \alpha'_n\} \cup \bigcup_{i=1}^n Q_i, (\Phi'_c - \{\alpha'_1, \dots, \alpha'_n\}) \cup \bigcup_{i=1}^n \Phi_{ci})$
- final states:  $Q'_f$
- transition rules:  $\Delta' \cup \bigcup_{i=1}^n \Delta_i \cup \bigcup_{i=1}^n \{q_{fi} \xrightarrow{\epsilon} \alpha'_i \mid q_{fi} \in Q_{fi}\}$



Example 9

An automaton for  $E[\text{var}]$  is:

$$\begin{array}{ll}
 q'_{E_1} \xrightarrow{\epsilon} q'_E & \alpha' \xrightarrow{\epsilon} q'_{E_1} \\
 q'_{E_2} \xrightarrow{\epsilon} q'_E & \text{app}(q'_E, q'_M) \rightarrow q'_{E_2} \\
 q'_{E_3} \xrightarrow{\epsilon} q'_E & \text{app}(q'_{E_4}, q'_E) \rightarrow q'_{E_3} \\
 \text{lam}(q'_{E_5}, q'_M) \rightarrow q'_{E_4} & \text{var} \rightarrow q'_{E_5} \\
 \\ 
 \text{var} \rightarrow q_{E_6} & q_{E_6} \xrightarrow{\epsilon} \alpha'
 \end{array}$$

where  $q'_E$  is the final state and  $\{q'_{E_1}, q'_{E_2}, q'_{E_3}\}$  is the conflict set. The transition rules for  $q'_M$  are omitted.

Note that the conflict sets are also copied because alternatives in a context definition remain alternatives when the hole is filled with a term.

From Context Definitions to Automata

General context definitions do not belong to the class of regular tree grammars.

For example, the context definition

$$E ::= b([\ ]) \mid a(E[c([\)])]$$

where  $a, b,$  and  $c$  are unary constructors defines contexts  $a^n(b(c^n([\ ])))$ . This language is not accepted by any regular tree grammar. It is however accepted by a *context-free tree grammar*. A context-free tree grammar is a tree grammar  $(\Sigma, N, S, P)$  where the rules have the form  $L(\alpha_1, \dots, \alpha_n) \rightarrow t$ , where  $L$  is a non-terminal and  $t \in \mathcal{T}(\Sigma \cup N \cup$

$\{\alpha_1, \dots, \alpha_n\}$ .

Example 10

A context-free tree grammar for  $a^n(b(c^n([\ ])))$  is

$$E(\alpha) \rightarrow b(\alpha) \mid a(E(c(\alpha)))$$

As expected, the equivalence and ambiguity of context-free tree grammars are undecidable [CDG<sup>+</sup>99]. Fortunately, context definitions are in a special subclass where the right-hand sides of production rules are restricted to  $\mathcal{T}(\Sigma \cup N, \Phi)$  which, with some further restrictions, can be transformed to productions for regular tree grammars. For instance, the productions for the evaluation contexts  $E$  of Example 7 are:

$$E(\alpha) \rightarrow \alpha \mid \text{app}(E(\alpha), M) \mid \text{app}(\text{lam}(\text{var}, M), E(\alpha))$$

If we regard  $E(\alpha)$  as a single non-terminal, the grammar becomes a regular tree grammar. We can then build an automaton for  $E(\alpha)$  as shown in Example 8.

The following shows how we transform context definitions:

1. Introduce a variable for the hole of each context. In this way, each context is associated with a variable. Then, replace all occurrences of the holes by the corresponding variables and replace all occurrences of unfilled contexts by filling the contexts with the corresponding variables. The production rules in the resulting

context-free tree grammar  $G$  are grouped as follows:

$$\begin{array}{ccc} id_1(\alpha_{11}, \dots, \alpha_{1k_1}) & \rightarrow & r_{11} \mid \dots \mid r_{1m_1} \\ \vdots & & \vdots \\ id_n(\alpha_{n1}, \dots, \alpha_{nk_n}) & \rightarrow & r_{n1} \mid \dots \mid r_{nm_n} \end{array}$$

2. In each right-hand side  $r_{ij}$ , search for occurrences of  $id_l(t_1, \dots, t_{k_l})$  that do not match any left-hand side. For each such occurrence add the following production rules:

$$id_l(t_1, \dots, t_{k_l}) \rightarrow r'_{l1} \mid \dots \mid r'_{lm_l}$$

where  $r'_{lh}$  is the substitution of  $\alpha_{l1}, \dots, \alpha_{lk_l}$  by  $t_1, \dots, t_{k_l}$  in  $r_{lh}$ .

3. Consider the left-hand side of each production rule as a single non-terminal and do the appropriate renaming.

This process, when it terminates (see next paragraph), produces a grammar  $G'$  that differs from a regular tree grammar only in allowing the subterms  $id_l(t_1, \dots, t_{k_l})$  in the right-hand sides of the production rules. Fortunately, we can build automata for these subterms by instantiating the abstract automata for  $id_l(\alpha_{l1}, \dots, \alpha_{lk_l})$ . Hence, we can build an automaton for the grammar  $G'$ .

Expansions in the second step may not terminate. To gain more intuition, consider the expansion of  $E(c(\alpha))$  in Example 10, which creates

$$E(c(\alpha)) \rightarrow b(c(\alpha)) \mid a(E(c(c(\alpha))))$$

where  $E(c(c(\alpha)))$  is to be expanded.

We have seen that constraints are needed for transforming context-free tree grammars. Given a context-free tree grammar  $(\Sigma, N, S, P)$ , for any recursive non-terminal application  $id_l(t_1, \dots, t_{k_l})$  in the right-hand side of a production rule  $id_i(\alpha_{i1}, \dots, \alpha_{ik_i}) \rightarrow r$ , each  $t_h$  for  $1 \leq h \leq k_l$  should be either a variable or a term in  $\mathcal{T}(\Sigma \cup N_i)$  where  $N_i$  is the set of non-terminals not recursively defined with  $id_i$ .

These constraints guarantee termination of the expansion of context-free tree grammars described. First, the arguments of recursive non-terminal applications do not contain recursive non-terminal applications, so that we do not expand arguments. Second, the arguments do not contain variables. When a group of production rules is instantiated, the arguments of recursive non-terminal applications in the resulting right-hand sides are either in the original right-hand sides or replacements for variables. In other words, all arguments in the generated production rule are either variables or arguments in the original context-free tree grammar. The combination of arguments can be exhausted since both sets of variables and original arguments used in the grammar are finite. Therefore, there is no infinite expansion.

The constraints are not necessary conditions for being able to build automata for contexts, but they are sufficient. Moreover, context definitions under our constraints are expressive enough for production rules in most grammars of interest. Most context definitions in programming language semantics satisfy these constraints.

### Polymorphism

A polymorphic type declaration defines a parametric type where the parameters are type variables. A monomorphic type has no parameter. We use state variables to represent type variables. The automata for polymorphic types are abstract au-

tomata as well and type instantiation corresponds to linking the state variables. We add one more component,  $\Phi_t$ , to the tuple representation of automata which is now  $(\Sigma, (Q, \Phi_t, \Phi_c), Q_f, \Delta)$  where  $\Phi_t$ ,  $\Phi_c$  are the sets of state variables for polymorphic types and contexts, respectively.

Polymorphic type definitions also belong to context-free tree grammars. Building automata for polymorphic types is similar to building automata for contexts and requires the same constraints.

For example, a polymorphic type *List* is defined as follows:

$$List(\alpha) \rightarrow nil \mid cons(\alpha, List(\alpha))$$

The automaton for the type `List` above is

$$nil \rightarrow q_{List} \quad cons(\alpha, q_{List}) \rightarrow q_{List}$$

where  $\alpha$  is the state variable for the type variable  $\alpha$ .

The automaton for the type `int List` is

$$\begin{array}{ll} nil \rightarrow q'_{List} & cons(\alpha', q'_{List}) \rightarrow q'_{List}, \\ q_{int} \rightarrow \alpha' & allextcept(int, \{\}) \rightarrow q_{int} \end{array}$$

where the automaton for the type *int* is linked to the state  $\alpha'$ .

### Automatically proving unique-decomposition in the SL System

To automatically check unique decomposition, we introduce a new phrase for SL specifications:

$$\text{decomp } T \text{ of } p_1 \mid \dots \mid p_n ; ;$$

where  $T$  is a type describing the terms and  $p_1, \dots, p_n$  are the patterns describing decompositions. The SL compiler checks the unique-decomposition phrase before generating the interpreter. The implementation follows the approaches presented in previous sections.

The output of checking includes the information about the existence of decompositions for terms in the type specified in the `decomp` phrase. The information about uniqueness is shown only when the existence is satisfied. Counterexamples are given when either fails.

For the call-by-value  $\lambda$ -calculus example, we augment the SL specification in Figure 4 with two dynamic definitions for redexes and faulty terms, and a `decomp` phrase. The `decomp` phrase consists of the type `M` and alternative patterns for the four decompositions: a value, a demand of a variable in an evaluation context, a redex in an evaluation context, and a faulty term in an evaluation context. The resulting specification is in Figure 9.

The SL compiler checks unique decomposition for the specification in Figure 9 and produces the following result:

---

All terms have decompositions.

---

 SIGNATURE:

```

type M = Num of int | Var of string
        | Lam of string*M | App of M*M;;

```

```

startfrom M;;

```

## SPECIFICATION:

```

... (* other parts of the specification, omitted *)

```

```

dynamic V = Num _ | Lam(_,_);;

```

```

dynamic R = App(Lam(_,_),V);;

```

```

dynamic F = App(Num _,_);;

```

```

context E = BOX | App(E, _) | App(V, E);;

```

```

decomp M of (_:V) | (_:E) (Var _)
            | (_:E) (_:R) | (_:E) (_:F);;

```

---

FIGURE 9. SL Specification of Decomposing a Simple CBV Language

Counterexamples to unique decomposition:

```
App((Num(_), Var(_)))
App((App((Num(_), Var(_))), Num(_)))
... (* more counterexamples omitted *)
```

---

The counterexamples provide hints about how to modify the specification in order to obtain uniqueness. Violating uniqueness originates from applying a number to another term. It can be fixed by restricting evaluation in the right subterm of an application to occur only when the left is a  $\lambda$ -abstraction. We modify the evaluation context in the specification in Figure 9 as follows:

---

```
context E = BOX | App(E, _) | App(Lam(_, _), E);;
```

---

Unique decomposition holds in the new specification. The output is:

---

```
All terms have unique decompositions.
```

---

The SL system does not enumerate all counterexamples. We have tested the SL system on some small examples, including the specification in Figure 9 and the specification for the example in Chapter I. The SL system requires only a few seconds for these



examples. When we extend the datatype in the specification with more constructors, the execution time increases dramatically. For example, if we add an addition operation to the specification shown in Figure 9, proving the unique-decomposition lemma takes about two minutes. Further extensions such as additional built-in types may require over an hour. Optimizing the algorithms and the implementation is one of our future goals.

### Summary and Discussion

Checking existence and uniqueness of decomposition is important for specifying operational semantics using syntactic theories. Existence ensures that the evaluators consider all the cases for an input program; uniqueness guarantees that evaluators are deterministic at each step. We have introduced algorithms for automatically proving unique-decomposition lemmas. These algorithms are implemented in the SL system. A number of small examples have been tested in the system.

Tree automata are widely used in compilation, especially in pattern matching. In terms of pattern matching, equivalence of tree automata corresponds to exhaustiveness of patterns and non-ambiguity of regular tree grammars corresponds to non-overlapping of patterns. However, the patterns used in earlier research do not support dynamic values and contexts.

Fähndrich and Boyland's work on abstract patterns [FB97] is very closer to our work. The difference is that the SL system provides counter examples when unique decomposition does not hold.

## CHAPTER IV

### EXTENSIONS TO THE SL SYSTEM

In this chapter, we enrich the SL system with more expressive power and flexibility to specify syntactic theories. The first extension is the notion of abstract pattern which is a natural extension to dynamic definitions and context definitions. The second extension is natural deduction, another common specification style for syntactic theories. The third extension is signature functions which specify computations on signature types. The last extension is meta-substitution which is implemented internally to provide general substitution utilities for all object-languages.

For each extension to the SL language, we also explain the corresponding extensions to the SL compiler with respect to the issues discussed in Chapter II such as type checking and pattern matching.

#### Abstract patterns

It is natural to extend from dynamic expressions and context expressions to abstract patterns which are introduced in Fähndrich and Boyland's paper [FB97].

We extend SL patterns with a new form called *abstract constraint patterns* (also called *abstract patterns* or *abspats* in short):  $(p : \text{abspat\_name})[p_2]$  where *abspat\_name* is the name of the defined abstract pattern and  $p_2$  is optional depending on whether the *abspat\_name* takes arguments. The left-hand side of an abspat definition is a name and optional arguments. The arguments are holes which are labeled so that they can be distinguished from each other. The right-hand side is a list of alternative patterns in which

the holes can be used. All arguments, *i.e.*, holes, need to occur once and only once in each alternative pattern.

The SL pattern definition becomes:

$$\begin{array}{l}
 \textit{Patterns} \qquad \qquad \qquad P ::= \dots \mid \\
 \qquad \qquad \qquad \qquad \qquad \qquad (P : \textit{abspat\_name}) [P] \\
 \textit{Abspat Definitions} \quad H(\square_1, \dots, \square_n) ::= P \mid \dots \mid P \quad \text{where } n \geq 0
 \end{array}$$

Dynamic definitions and context definitions are special cases of abstract pattern definitions for which  $n$  is either 0 or 1, respectively.

In the SL language syntax, an abstract pattern definition starts with `abspat`, followed by a list of variables and a list of pattern alternatives. Figure 10 shows how to use abstract patterns to rewrite the example given in Figure 4. The dynamic value declaration for `V` and the context declaration for `E` are replaced with abstract pattern definitions.

Abstract patterns can be used to easily specify semantic operations. The following axiom describes swapping the head and tail of a list with at least two elements by using abstract patterns.

---

```

abspat last(t) = [t] | _::((_:last)(t));
abspat firstlast(h,t) = h::((_:last)(t));
axiom swap_firstlast: (l:firstlast)(h,t) ==> l(t, h);

```

---

The abstract pattern `last` describes a non-empty list with a hole for its last element. The abstract pattern `(l:firstlast)(h,t)` matches a list with at least two

---

SIGNATURE:

```
type M = Var of string | Lam of string*M | App of M*M;;
startfrom M;;
```

SPECIFICATION:

```
#open "namesupply";;
let rec subst (t1,x,t2) =
  match t1 with
  | Var s -> if s = x then t2 else t1
  | Lam(s,t1') -> if s = x then t1
    else let s' = freshname() in
      Lam(s', subst(subst(t1',s,Var s'),x,t2))
  | App(t11,t12) -> App(subst(t11,x,t2),subst(t12,x,t2));;
```

```
abspat V = Lam _;;
```

```
axiom betav: App(Lam(x,t1), (t2:V)) ==> subst(t1, x, t2);;
```

```
abspat H(x) = x | App((_:H)(x),_) | App((_:V),H);;
```

inference eval:

```
t1 ==> t2
```

```
-----
(h:H) t1 |==> h t2 ;;
```

---

FIGURE 10. An SL Specification of a Simple CBV Language using abstract patterns

elements. It binds `l` to a list with two holes occurring in the first position and the last position. It also binds `h` and `l` to the first element and the last element, respectively. If a list is `[1;4;3;2;5]`, then pattern `(l:headtail) (h, t)` can be matched successfully with `l` bound to a function `[□;4;3;2;□]`, `h` bound to 1, and `t` bound to 5. Therefore, the result of reducing the list `[1;4;3;2;5]` by axiom `swap_firstlast` becomes `[5;4;3;2;1]`.

Similarly, we can use abstract patterns to specify a simple sorting axiom:

---

```

abspat anyone(e) = e::_ | _::((_:anyone) (e));
abspat anytwo(e1,e2) = e1::((_:anyone) (e2))
                    | _::((_:anytwo) (e1,e2));
axiom swap: (l:anytwo) (e1,e2) when e1 > e2 ==> l(e2,e1);

```

---

The sorting swaps two elements in the list if the prior element is greater than the latter. Picking two element is non-deterministic. A possible reduction sequence for the list `[1;4;3;2;5]` is the following:

---

```

[1;4;3;2;5] ==> (* swap 4 and 3 *)
[1;3;4;2;5] ==> (* swap 4 and 2 *)
[1;3;2;4;5] ==> (* swap 3 and 2 *)
[1;2;3;4;5]

```

---

Another possible sequence could be:

---

```
[1;4;3;2;5] ==> (* swap 4 and 2 *)
[1;2;3;4;5]
```

---

Next, we will explain the extensions to type checking and pattern matching for abstract patterns.

### 1. Type checking for abstract patterns

The typing rules for abstract pattern are given in Table 7. The rule for non-parametric abstract constraint patterns is similar to the rule for dynamic constrain patterns. The type of a non-parametric abstract constraint pattern is the same as the type of the abstract pattern and this type cannot be a context type. The rule for parametric abstract constraint patterns is similar to the rule for context constraint patterns. The type of a parametric abstract pattern is a context type  $\tau_1 * \dots * \tau_n \circ \rightarrow \tau$ , where  $\tau_1, \dots, \tau_n$  are the types for the holes and  $\tau$  is the resulting type when all holes are filled. The type of the corresponding parametric abstract constrain pattern is either the resulting type of the abstract pattern if the pattern to be filled is not a context type, or the composition of context types if the pattern to be filled is a context type. One new rule is added for tuple patterns when there is more than one component having a context type. The rule lifts the context-type constructor to the top level.

The typing rules for abstract pattern definitions are given in Table 8. The rules are similar to the rules for dynamic definitions and context definitions. All alternative patterns should have the same type as the abstract pattern.

$\frac{\Gamma \vdash p : \tau}{\Gamma, ap : \tau \vdash (p : ap) : \tau}$
$\frac{\Gamma \vdash p_1 : \tau_0 \circ \tau_1 \quad \Gamma \vdash p_2 : \tau_0}{\Gamma, ap : \tau_0 \circ \tau_1 \vdash (p_1 : ap)[p_2] : \tau_1}$
$\frac{\Gamma \vdash p_1 : \tau_1 \circ \tau_2 \quad \Gamma \vdash p_2 : \tau_0 \circ \tau_1}{\Gamma, ap : \tau_1 \circ \tau_2 \vdash (p_1 : ap)[p_2] : \tau_0 \circ \tau_2}$
$\frac{\Gamma \vdash p_1 : \tau_1 \quad \Gamma \vdash p_i : \sigma_i \circ \tau_i \quad \Gamma \vdash p_j : \sigma_j \circ \tau_j \quad \Gamma \vdash p_n : \tau_n}{\Gamma \vdash (p_1, \dots, p_i, \dots, p_j, \dots, p_n) : \sigma_i * \sigma_j \circ \tau_1 * \dots * \tau_n}$

TABLE 7. The type checking rules for abstract patterns

$\Gamma, ap : \tau \vdash p_1 : \tau$
$\vdots$
$\Gamma, ap : \tau \vdash p_n : \tau$
$\Gamma, ap : \tau \vdash ap = p_1 \mid \dots \mid p_n : \tau$
$\Gamma, ap : (\sigma_1 * \dots * \sigma_m) \circ \rightarrow \tau \vdash p_1 : (\sigma_1 * \dots * \sigma_m) \circ \rightarrow \tau$
$\vdots$
$\Gamma, ap : (\sigma_1 * \dots * \sigma_m) \circ \rightarrow \tau \vdash p_n : (\sigma_1 * \dots * \sigma_m) \circ \rightarrow \tau$
$\Gamma, ap : (\sigma_1 * \dots * \sigma_m) \circ \rightarrow \tau \vdash ap(\square_1, \dots, \square_m) = p_1 \mid \dots \mid p_n : (\sigma_1 * \dots * \sigma_m) \circ \rightarrow \tau$

TABLE 8. The type checking rules for abspat definitions

## 2. Pattern matching for abstract patterns.

Extending the pattern matching algorithm is straightforward since matching abstract patterns is similar to matching dynamic values or matching context expressions.

After preprocessing and splitting the matching matrix, we need to consider a new group in addition to the existing variable and construct groups. All the first column patterns in the group are abspat constraint patterns with the same abstract pattern.

For an abspat constraint pattern group, we consider two subcases for pattern matching function  $\mathcal{C}$ .

### – Non-parametric abstract patterns

Non-parametric abstract patterns are similar to dynamic values. The function  $\mathcal{C}$  creates a reference state. The reference will initiate pattern matching for



the abstract pattern  $P$  with the value of  $t_1$ . Its result is bound to a new parameter. The state in the let body is the result of compiling a structure consisting of the patterns without the constraint and the rest of the patterns.

$$\mathcal{C} \left( \begin{array}{c} t_1 \quad \cdots \\ (p'_{11} : P) \cdots s_1 \\ \vdots \quad \cdots \quad \vdots \\ (p'_{m1} : P) \cdots s_m \end{array} \right) \longrightarrow \text{let } t'_1 = \text{match\_}P \ t_1 \text{ in } \mathcal{C} \left( \begin{array}{c} t'_1 \cdots \\ p'_{11} \cdots s_1 \\ \vdots \quad \cdots \quad \vdots \\ p'_{m1} \cdots s_m \end{array} \right)$$

#### – Parametric abstract patterns

Parametric abstract patterns are similar to context expressions. The function  $\mathcal{C}$  creates a reference state. The reference will initiate pattern matching for the context definition  $P$  with the value of  $t_1$ . Its result is bound to a pair of parameters which represent the abstract pattern and the corresponding holes occurring in  $t_1$ . The state of the let body is the result of compiling a structure consisting of the context patterns and the hole patterns, as well as the rest of the patterns.

$$\begin{array}{c}
 \text{(evalv)} \quad \frac{}{v \twoheadrightarrow v} \\
 \\
 \text{(evalm)} \quad \frac{t_1 \twoheadrightarrow \lambda x. t_3 \quad t_2 \twoheadrightarrow v}{t_1 t_2 \twoheadrightarrow t_3[x := v]}
 \end{array}$$

FIGURE 11. call-by-value lambda calculus in natural deduction semantics

$$C \left( \begin{array}{c} t_1 \quad \dots \\ (p'_{11} : P)p''_{11} \dots s_1 \\ \vdots \quad \dots \quad \vdots \\ (p'_{m1} : P)p''_{m1} \dots s_m \end{array} \right) \rightarrow \text{let } (t'_1, t''_1) = \text{match\_H } t_1 \text{ in } C \left( \begin{array}{c} t'_1 \quad t''_1 \quad \dots \\ p'_{11} \quad p''_{11} \dots s_1 \\ \vdots \quad \vdots \quad \dots \quad \vdots \\ p'_{m1} \quad p''_{m1} \dots s_m \end{array} \right)$$

### Natural Deduction Rules

Natural deduction is an important way to write syntactic theories. The operational semantics of many languages and as well as most type systems are described with natural deduction rules. Figure 11 shows evaluation rules for the call-by-value lambda calculus, where  $v$  stands for values and  $\twoheadrightarrow$  stands for zero or more steps. The rule *evalv* states that values are normal forms. The rule *evalm* states that for an application  $t_1 t_2$ , if  $t_1$  can be evaluated in zero or more steps to a function  $\lambda x. t_3$  and  $t_2$  can be evaluated to a value  $v$ , then the whole application can be evaluated to a term by substituting free occurrences of  $x$  in  $t_3$  with the value  $v$ .

We extend the SL language to allow multiple premises in inference rules. Each premise has an expression as its left-hand side and a pattern for its right-hand side. The

new definition for inference rules is as follows:

$$\text{Inference Rules } I ::= \text{inf\_name} : \frac{E \implies P \quad \dots \quad E \implies P}{P \text{ when } E \implies E}$$

The call-by-value lambda calculus in Figure 4 can be re-stated as in Figure 4.2 using natural deduction rules. The inference rules `veval` and `meval` correspond to the deduction rules `evalv` and `evalm`, respectively.

Next, we explain the extensions to the SL compiler for type checking and pattern matching.

### 1. Type checking

The typing rule for the extended inference rule is given in Table 9. The extension to the typing rules is straightforward. The only change is to consider multiple premise clauses in inference rules. Both sides of each premise need to have the same type. Different premise clauses may have different types.

### 2. Pattern matching

The pattern matching algorithm also needs to consider multiple premises in an inference rule. This is done by using different continuation functions when matching premises.

In the pattern matching algorithm, the initialization for inference becomes:

- For an inference rule

$$\frac{e_1 \implies p_1, e_2 \implies p_2, \dots, e_n \implies p_n}{p \text{ when } e_c \implies e},$$

---

```

(* Example: call-by-value lambda calculus with natural
   deduction *)
SIGNATURE:
type M = Var of string | Lam of string*M | App of M*M;;
startfrom M;;

SPECIFICATION:
#open "namesupply";;
let rec subst (t1,x,t2) = ...;;

dynamic V = Lam _;;

inference meval:
t1 ==>> Lam(x,t1'); t2 ==>> (t2':V)
-----
App(t1,t2) ==>> subst(t1',x,t2')
;;

inference veval:
-----
(v:V) ==>> v
;;

inference Eapp1:
t1 ==>> t1'
-----
App(t1,t2) ==>> App(t1',t2)
;;

inference Eapp2:
t2 ==>> t2'
-----
App((t1:V),t2) ==>> App(t1,t2')
;;

```

---

FIGURE 12. An SL Specification of a Simple CBV Language specified with natural deduction rules

$\Gamma, \dots, x_i : \rho_i, \dots \vdash p : \tau$
$\Gamma, \dots, x_i : \rho_i, \dots \vdash e_c : \text{bool}$
$\Gamma, \dots, x_i : \rho_i, \dots \vdash e_1 : \sigma_1$
$\Gamma, \dots, x_i : \rho_i, \dots, y_{1i} : \rho_{1i}, \dots \vdash p_1 : \sigma_1$
$\vdots$
$\Gamma, \dots, x_i : \rho_i, \dots \vdash e_n : \sigma_n$
$\Gamma, \dots, x_i : \rho_i, \dots, y_{ni} : \rho_{ni}, \dots \vdash p_n : \sigma_n$
$\Gamma, \dots, x_i : \rho_i, \dots, y_{1i} : \rho_{1i}, \dots, y_{ni} : \rho_{ni} \vdash e : \tau$
<hr style="width: 50%; margin: 0 auto;"/>
$\Gamma \vdash \text{inf\_name} : \frac{e_1 ==> p_1 \quad \dots \quad e_m ==> p_m}{p \text{ when } e_c ==> c} : \tau$

TABLE 9. The type checking rules for extended inference rules

the corresponding state is:

```

    if  $e_c$  then
      let  $p_1 = \text{rewrite1 } e_1$  in
      let  $p_2 = \text{rewrite1 } e_2$  in
      :
      let  $p_n = \text{rewrite1 } e_n$  in accept  $e$ 
    else fail
  
```

where `rewrite1` is a one-step rewriting function in the generated code. To accept the final expression  $e$ , the pattern  $p$  needs to be matched and the condition  $e_c$  needs to be true. In addition, all the premise clauses need to be satisfied by rewriting the left-hand side expressions successfully and binding the results to the right-hand side patterns successfully. Any failure in rewriting the left-hand side expressions in the premise clauses will cause failure in rewriting with the inference rule.

### Signature Functions

As we have seen, CAML functions are often used in the SPECIFICATION part. Although these functions can specify computations in object-languages, they are given with rules. As a result, the computation in these functions is not easy to track because the SL system treats CAML function calls as atomic actions.

To address this issue, we introduce signature functions (*sigfuns* for short) to specify computation with SL signature datatypes. All the argument and the result types should be SL signature types. In other words, signature functions are functions handling



---

SIGNATURE:

```
type M = Var of string | Lam of string*M | App of M*M;;
startfrom M;;
```

SPECIFICATION:

```
#open "namesupply";;
```

```
sigfun subst (t1,x,t2) =
  (Var s,_,_) ==> if s = x then t2 else t1
| (Lam(s,t1'),_,_) ==> if s = x then t1
  else let s' = freshname() in
  Lam(s', subst(subst(t1',s,Var s'),x,t2))
| (App(t11,t12),_,_) ==> App(subst(t11,x,t2),subst(t12,x,t2));;
```

```
absPAT V = Lam _;;
```

```
axiom betaV: App(Lam(x,t1), (t2:V)) ==> subst(t1, x, t2);;
```

```
absPAT H(x) = x | App((_:H)(x),_) | App((_:V),H);;
```

inference eval:

```
t1 ==> t2
```

```
-----
```

```
(h:H) t1 |==> h t2 ;;
```

---

FIGURE 13. An SL Specification of a Simple CBV Language using signature functions



```

match env with
  [] -> raise Not_found
| (s1,v)::rest ->
    if (s == s1) then v else getenv (rest, s)

```

---

The function can be rephrased with an abstract pattern and a sigfun definition. The abspat `firstoccur` defines the first binding of the name `s` in an environment and binds the corresponding value to `v`. Sigfun `getenv` matches the `firstoccur` with `env` and returns the corresponding value if it matches. If it does not match, a predefined exception `rewrite_failure` will be raised.

---

```

abspat firstoccur(s,v) = (s,v)::_
  | (s1,v1)::((_:firstoccur) (s,v)) when s1 != s

sigfun getenv
  (((_:firstoccur) (s1,v)) as env, s) when s = s1 => v;;

```

---

The introduction of signature function definitions requires minor extensions to the SL compiler. We explain the extensions next.

(a) Type checking for sigfun definitions

The typing rules for signature function definitions are given in Table 10.

The rules are similar to the typing rules for function definitions in CAML.

$\begin{array}{l} \Gamma, x_1 : \tau_1, \dots, x_m : \tau_m, \dots, y_{1i} : \rho_{1i}, \dots \vdash p_1 : \tau_1 * \dots * \tau_m \\ \Gamma, x_1 : \tau_1, \dots, x_m : \tau_m, \dots, y_{1i} : \rho_{1i}, \dots \vdash e_1 : \tau \\ \vdots \\ \Gamma, x_1 : \tau_1, \dots, x_m : \tau_m, \dots, y_{ni} : \rho_{ni}, \dots \vdash p_n : \tau_1 * \dots * \tau_m \\ \Gamma, x_1 : \tau_1, \dots, x_m : \tau_m, \dots, y_{ni} : \rho_{ni}, \dots \vdash e_n : \tau \end{array}$ <hr style="width: 80%; margin: 10px auto;"/> $\Gamma \vdash \text{sigfun } sf \ p_1 \Rightarrow e_1 \mid \dots \mid p_n \Rightarrow e_n : \tau_1 * \dots * \tau_m \rightarrow \tau$
---

TABLE 10. The type checking rules for sigfun definitions

All alternative patterns need to be the same type which is the same as the argument type for the signature function. All resulting expressions need to be the same type which is also the result type for the signature function.

(b) Pattern matching for sigfun definitions

Pattern-matching for signature functions uses the same form of structures and uses the same algorithm described in Chapter II. Given a signature function definition:

$$\begin{array}{l} \text{sigfun } f(x_1, \dots, x_m) \ p_1 \Rightarrow e_1 \mid \\ \dots \\ p_n \Rightarrow e_n \end{array}$$



rules. First, `subst` is declared as a signature function mapping a tuple to the object-type. Two sigfun axioms and one sigfun inference rule correspond to the three alternatives in the sigfun definition in Figure 13.

Compiling rule-style sigfun definitions is similar to compiling semantic rules. We discuss the extensions to the SL compiler next.

(a) Type checking for sigfun rules

Sigfuns must be declared with explicit types before it is used in sigfun axioms and sigfun inference rules. The typing rule for sigfun declarations is straightforward. The type of the sigfun is as declared. The rule is given below.

$$\frac{\Gamma \vdash te_1 : \sigma \quad \Gamma \vdash te_2 : \tau}{\Gamma \vdash \text{sigfun } sf : te_1 - > te_2 : \sigma \rightarrow \tau}$$

The typing rules for sigfun rules are similar to those for axioms and inference rules. The rule for sigfun axioms is given in Table 11. Both sides of a sigfun axiom should have the same type and the type cannot be a context type. The condition expression in an axiom should be of boolean type. A new typing rule needs to be given for applying a signature function to a pattern. This rule is similar to the rule for function application expressions. If a signature function  $sf$  has type  $\tau_1 \rightarrow \tau_2$  and a pattern  $p$  has type  $\tau_1$ , then the application  $sf(p)$  has type  $\tau_2$ .

Similarly, both sides of each clause in a sigfun inference rule should have the

---

```

SIGNATURE:
type M = Var of string | Lam of string*M | App of M*M;;
startfrom M;;

SPECIFICATION:
#open "namesupply";;

sigfun subst : M*string*M -> M;;

axiom substVar:
subst((Var s) as t1, x,t2) ==> if s = x then t2 else t1;;

axiom substLam:
subst((Lam(s,t1')) as t1, x, t2) ==>
  if s = x then t1
  else let s' = freshname() in
        Lam(s', subst(subst(t1',s,Var s'),x,t2))
;;

inference substApp:
subst(t11, x, t2) ==> t11', subst(t12, x, t2) ==> t12'
-----
subst(App(t11,t12), x, t2) -> App(t11',t12')
;;

abspat V = Lam _;;

axiom betav: App(Lam(x,t1), (t2:V)) ==> subst(t1, x, t2);;

abspat H(x) = x | App((_:H)(x),_) | App((_:V),H);;

inference eval:
t1 ==> t2
-----
(h:H) t1 |==> h t2 ;;

```

---

FIGURE 14. An SL Specification of a Simple CBV Language using signature functions

$\frac{\Gamma, sf : \sigma \rightarrow \tau \vdash sf : \sigma \rightarrow \tau \quad \Gamma, x_1 : \tau_1, \dots, x_n : \tau_n \vdash p : \sigma}{\Gamma, sf : \sigma \rightarrow \tau \vdash sf(p) : \tau}$
$\Gamma, sf : \sigma \rightarrow \tau, x_1 : \tau_1, \dots, x_n : \tau_n \vdash sf(p) : \tau$
$\Gamma, sf : \sigma \rightarrow \tau, x_1 : \tau_1, \dots, x_n : \tau_n \vdash e_c : bool$
$\frac{\Gamma, sf : \sigma \rightarrow \tau, x_1 : \tau_1, \dots, x_n : \tau_n \vdash e : \tau}{\Gamma, sf : \sigma \rightarrow \tau \vdash ax\_name : sf(p) \text{ when } e_c ==> e : \tau}$

TABLE 11. The type checking rules for sigfun axioms

same type. This type cannot be a context type and the clause itself is considered to have that type as well. All the premise clauses and the conclusion clause need not necessary have the same type. The condition expression in a sigfun inference rule should have boolean type. The rule for sigfun inference rules is given in Table 12. In the table,  $\Delta$  denotes the environment for all defined sigfuns:

$$\Delta = \{sf : \sigma \rightarrow \tau, \dots, sf_i : \sigma_i \rightarrow \tau_i, \dots\}$$

(b) Pattern matching for sigfun rules

Pattern matching for sigfun rules uses the same form of structures and uses the same algorithm as used in Chapter II. Each signature function corresponds to a pattern-matching structure. By compiling the structure, an automaton and matching functions in CAML can be generated.

$\Gamma, \Delta, \dots, x_i : \rho_i, \dots \vdash p : \sigma$
$\Gamma, \Delta, \dots, x_i : \rho_i, \dots \vdash e_c : \text{bool}$
$\Gamma, \Delta, \dots, x_i : \rho_i, \dots \vdash e_1 : \sigma_1$
$\Gamma, \Delta, \dots, x_i : \rho_i, \dots, y_{1i} : \rho_{1i}, \dots \vdash p_1 : \tau_1$
$\vdots$
$\Gamma, \Delta, \dots, x_i : \rho_i, \dots \vdash e_n : \sigma_n$
$\Gamma, \Delta, \dots, x_i : \rho_i, \dots, y_{ni} : \rho_{ni}, \dots \vdash p_n : \tau_n$
$\Gamma, \Delta, \dots, x_i : \rho_i, \dots, y_{1i} : \rho_{1i}, \dots, y_{ni} : \rho_{ni} \vdash e : \tau$
$\Gamma, \Delta \vdash \text{inf\_name} : \frac{sf_1(e_1) ==> p_1 \quad \dots \quad sf_m(e_m) ==> p_m}{sf(p) \text{ when } e_c ==> c} : \tau$

TABLE 12. The type checking rules for sigfun inference rules

First, all sigfun rules for the same signature function are collected. The rules for a signature function can be:

- Sigfun axioms whose left-hand sides are applications of the signature function.
- Sigfun inference rules whose conclusion clauses have applications of the signature function on the left-hand side.

Second, the signature functions are stripped. As a result, the sigfun applications to patterns become patterns.

Finally, the pattern-matching algorithm described in Chapter II can be used. The treatment of sigfun axioms and sigfun inference rules is almost the same as that for axioms and inference rules. The only difference is in the initialization of the matching matrix for sigfun inference rules.

- For a sigfun inference

$$\frac{sf_1(e_1) ==> p_1, sf_2(e_2) ==> p_2, \dots, sf_n(e_n) ==> p_n}{sf(p) \text{ when } e_c ==> e},$$

the corresponding state is:

```

if  $e_c$  then
    let  $p_1 = match\_sf_1\ e_1$  in
    let  $p_2 = match\_sf_2\ e_2$  in
    :
    let  $p_n = match\_sf_n\ e_n$  in accept  $e$ 
else fail

```



where  $match\_sf_i, i = 1, 2, \dots, n$  are the matching functions for signature functions  $sf_1, sf_2, \dots, sf_n$ , respectively. To accept the final expression  $e$ , the pattern  $p$  needs to be matched with the argument of the signature function  $sf$  and the condition  $e_c$  needs to be true. In addition, all the premise clauses needs to be satisfied. In other words, for each premise,  $sf_i(e_i)$  need to successfully match the definition of the signature function  $sf_i$  and the result must be successfully bound to  $p_i$ . Any failure in the above will cause failure in applying the signature function.

### Meta-substitution

In the previous SL examples, we have used a substitution function written in CAML. The substitution function needs to be changed for almost every language. It would be better to provide a meta-level substitution that can work for all languages.

We introduce a variable phrase to specify the binding of a free variable in the object-language. We also provide a meta-level function `sp_subst` to use for all substitutions, as well as a function `sp_fvars` to get the free variables in the object-language.

A variable phrase can be specified in the SIGNATURE part. It describes which constructor denotes free variables and which constructor introduces new bound variables, and what the scope of a new variable is. For example, we can extend the SIGNATURE part of the SL specification in Figure 4 with the following phrase:

---

```
variable x : Var x : Abs(x, scope) ; ;
```

---

This phrase describes the variables and bindings in lambda terms. For a name  $x$ ,

`Var x` stands for a free variable  $x$  and `Abs (x, scope)` means that `Abs` introduces a binding of variable  $x$ . The keyword `scope` represents the scope of variable  $x$ .

The meta-function `sp_subst` performs substitution in all cases if the variable bindings are provided. Like ordinary substitution, `sp_subst` takes three arguments: an expression to substitute, a variable name, and an object-expression to be substituted. The meta-function searches for all free occurrences of the variable with given name and replaces them with the given expression. The search is a depth-first traversing of the term to identify the occurrences of the specified variable constructors. For the constructor introducing bound variables, preprocessing is performed to ensure that names of bound variables are distinct from the name of the argument for `sp_subst`. Similarly, the meta-function `sp_fvars` returns free variables occurring in the given object-term.

The example given in Figure 15 uses the meta-function `sp_subst` instead of the CAML substitution function `subst`. This makes the SL specification more abstract and concise. Another advantage of using meta-substitution is that SL can track substitution which it cannot do when using substitution functions specialized for each object-language.

Another example shows the variable phrases used for the `Let` and `Letrec` constructs:

---

```

type T = Var of string | Abs of string * T | App of T* T |
        Let of string*T*T | Letrec of string*T*T
;;

variable x : Var x : Abs(x,scope);;
```

---

```

(* Example: call-by-value lambda calculus,
   using meta-substitutions *)

SIGNATURE:
type T = Var of string | Abs of string * T | App of T* T;;

variable x : Var x : Abs(x,scope);;

startfrom T;;

SPECIFICATION:
dynamic Value = Abs _;;

axiom beta:
App(Abs(x, t1), (t2 : Value)) ==>>
sp_subst (t1, x, t2)
;;

context E = BOX | App(E, _) | App(Value, E);;

inference eval:
t1 ==> t2
-----
(e : E) t1 ==> (e t2)
;;

```

---

FIGURE 15. An SL Specification of a Simple CBC Language with meta-substitution

```
variable x : Var x : Let(x,_, scope) ;;  
variable x : Var x : Letrec(x, scope, scope) ;;
```

---

Note that there can be more than one variable phrase for the same data-type and there can be more than one `scope` in a variable phrase. In the example above, `Abs`, `Let`, and `Letrec` all introduce bound variables and the variable constructor is `Var`. The `scope` occurs twice in the variable phrase for `Letrec`. The combination of the two `scopes` specifies the scope of variable `x` for `Letrec`.

There are two limitations of the meta-substitution function. First, a variable must use a string for its name. Second, a variable and its scope are introduced in the same constructor. However, these are limitations shared by all common languages with variable bindings.

### Summary and Discussion

By using the extensions described in this chapter, namely, abstract patterns, natural deductions, sigfunctions, and meta-substitutions, the SL system becomes more powerful for specifying syntactic theories and checking properties, such as subject reduction.

Abstract patterns are natural extensions from dynamic values and contexts. They make it very easy to abstract computations with patterns. Earlier work [FB97] on abstract patterns did not provide a mechanism on compile abstract patterns. We extend the pattern matching algorithm for dynamic values and contexts so that it applies to abstract patterns as well.

Natural deduction is another common style to represent operational semantic for programming languages. The support of natural deduction enables the SL system's

ability to specify another large range of language specifications.

The concept of sigfunctions is the result of combining function declarations and SL patterns as well as the rules. Definition of sigfunctions can be much more concise than the function definitions in CAML.

Meta-substitution is an important feature in the SL system, which make it much easier to specify substitution in syntact theories. Instead of writing tedious substitution function, it is enough now to specify the introduction of scope and free variables. Also, meta-substitution makes it very easy to check equivalence for the first-order terms, which will be discussed in the next Chapter.

## CHAPTER V

## AUTOMATICALLY PROVING SUBJECT REDUCTION

When designing a programming language, one important task is to design its type system. The main purpose of this task is to identify ill-behaved programs. For example, type systems can detect errors such as using a non-function type expression as function.

Designing a type system is a complicated task. Is the language strongly typed or weakly typed? Is the language explicitly typed or implicitly typed? One needs to carefully consider type constructs and their relationship with language constructs, so that the type system satisfies a number of important properties. Some other common questions about the properties of a type system are: Is it deterministic to derive a type from a term? Is it possible to derive two different types for a term? Does the type change when evaluating a term?

We focus on this last property in this chapter which is usually formalized as the subject reduction lemma. Proving the lemma is also laborious and error-prone. Several iterations are usually needed to tune the type system. Therefore, it is natural to consider an automatic way of proving the subject reduction.

We introduce a meta-layer in the SL system. The meta-layer consists of meta-expressions, meta-theorems, and meta-utilities. The subject reduction lemma is represented as a meta-theorem. Automatic induction is done by mechanically simplifying the meta-theorem. The simplification involves not only evaluation over meta-expressions, but also instantiation with rewriting rules, typing rules, or structure of data-types of object terms. The instantiation is equivalent to case analysis in an inductive proof. The

process is repeated until the simplified meta-theorem can be easily proved, for example, when it becomes a logical tautology.

The remainder of this chapter is organized as follows: We first show a manual proof of subject reduction for the simply-typed lambda-calculus. Then we give the specification of the typing rules for call-by-value lambda calculus in the SL system. Next we describe extensions with the meta-layer of the SL system and a new phrase to represent the subject-reduction lemma. Later we elaborate on the mechanism used to automatically prove the subject reduction lemma by induction and close with a summary.

### A Manual Proof of Subject Reduction

#### Lemma 4

Generation lemma:

1. if  $\Gamma \vdash x : \sigma$ , then  $(x : \sigma) \in \Gamma$ .
2. if  $\Gamma \vdash MN : \tau$ , then there exists a type  $\sigma$  such that  $\Gamma \vdash M : (\sigma \rightarrow \tau)$  and  $\Gamma \vdash N : \sigma$ .
3. if  $\Gamma \vdash \lambda x.M : \rho$ , then there exist types  $\sigma$  and  $\tau$  such that  $\Gamma, x : \sigma \vdash M : \tau$  and  $\rho \equiv (\sigma \rightarrow \tau)$ .

**Proof:** By induction on the length of derivation. □

#### Lemma 5

Substitution lemma:

1. if  $\Gamma \vdash M : \sigma$ , then  $\Gamma[\alpha := \tau] \vdash M : \sigma[\alpha := \tau]$
2. If  $\Gamma, x : \sigma \vdash M : \tau$ , and  $\Gamma \vdash N : \sigma$ , then  $\Gamma \vdash M[x := N] : \tau$ .

**Proof:**

1. By induction on the derivation of  $M : \sigma$ .
2. By induction on the generation of  $\Gamma, x : \sigma \vdash M : \tau$ .

□

### Lemma 6

Subject reduction lemma: Suppose  $M \rightarrow_{\beta} M'$  and  $\Gamma \vdash M : \sigma$ , then  $\Gamma \vdash M' : \sigma$ .

**Proof:** Induction on the generation of  $\rightarrow_{\beta}$  using Lemma 4 and Lemma 5.

We focus on the base case, namely that  $M \equiv (\lambda x.P)Q$  and  $M' \equiv P[x := Q]$ . If

$$\Gamma \vdash (\lambda x.P)Q : \sigma$$

then by Lemma 4

$$\Gamma, x : \tau \vdash P : \sigma \text{ and } \Gamma \vdash Q : \tau$$

and therefore by the substitution lemma 5,

$$\Gamma \vdash P[x := Q] : \sigma$$

□

### Representing the Type System in SL

The SL code for the call-by-value typed lambda calculus is shown in Figure 16.

The type of lambda terms is given in the SIGNATURE part in a datatype declaration. The types of lambda terms can be either type variables, “TVar s”, where s is a



string name, or function types, “TArrow(t1,t2)”, where “t1” is the argument type and “t2” is the result type.

In Figure 16, the sig-function “Typeof” is declared. The function takes a pair of a lambda term and a type environment as argument, and returns the term type. The type environment is a map from string names to types.

The typing rules are given by a sigfun axiom and two sigfun inference rules. The sigfun axiom phrase, given name “Vartype”, describes the base case of looking in the environment for type variables. The sigfun inference phrases, “Abstype” and “Apptype”, correspond to the typing rules for abstraction and application shown in Figure 1.

### A Meta-layer for the SL System

We introduce a meta-layer for the SL System. The meta-layer includes *meta-expressions*, *logic-expressions*, *meta-theorems*, and operations on them.

Meta-expressions extend the intermediate SL expressions with meta-variables.

Logic expressions are first-order logic formula that consists of meta-expressions as primitive expressions and logic operations. Logic operations include logical operators, such as logical “or” and “and”, and predicates. The predicates in the SL meta-layer are extended beyond the regular relational operations with semantic predicates to capture, for example, a reduction relation between two expressions or whether an expression matches an abstract pattern.

Meta-theorems are built from logic-expressions. Each meta-theorem has a list of meta-variables, a list of logic expressions as premises, and a list of logic expressions as conclusions.

The formal definitions of meta-expressions, logic-expressions and meta-theorems

---

```

(* Example: call-by-value typed lambda calculus *)
SIGNATURE:
type M = Var of string | Abs of string * M | App of M * M;;
type T = TVar of string | TArrow of T * T;;

startfrom M;;

SPECIFICATION:
dynamic Value = Abs _;;
context E = BOX | App(E, _) | App(Value, E);;

axiom beta:
App(Abs(x, t1), (t2 : Value)) ==>> sp_subst(t1, x, t2);;

inference eval:
t1 ==> t2
-----
(e : E) t1 ==> (e t2);;

fun Typeof : (M*((string*T) list)) -> T;;

axiom Vartype:
Typeof(Var x, ((y,TVar tau)::_)) when x=y ==> TVar tau;;

inference AbsType:
Typeof(e, ((x,TVar tau1)::gamma)) ==> tau2
-----
Typeof (Abs(x, e), gamma) ==> TArrow(tau1, tau2)
;;

inference AppType:
Typeof(e1,gamma) ==> TArrow(tau1,tau2);
Typeof(e2,gamma) ==> tau3
-----
Typeof (App(e1,e2), gamma) when tau1 = tau3 ==> tau2
;;

```

---

FIGURE 16. The SL specification of the type system of CBV  $\lambda$ -calculus

<i>Logic-Expressions</i>	$L ::= M \mid M = M \mid M < M \mid M \Rightarrow M \mid M \Rightarrow\Rightarrow M \mid$ $\text{mpat } M \ M \mid$ $L \text{ and } L \mid L \text{ or } L \mid \text{not } L \mid$
<i>Meta-Expressions</i>	$M ::= x \mid \sigma \mid \text{const} \mid c_0 \mid c_1 M \mid (M, \dots, M) \mid$ $\text{op}(M, \dots, M) \mid (M : \text{type } T_E) \mid$ $\lambda x. M \mid M \ M \mid M [M]$ $\text{let } P = M \text{ in } M \mid$ $\text{letrec } P = M \text{ in } M$
<i>Meta-Theorems</i>	$L ::= \frac{\text{thm\_name} : \text{forall } x_1, \dots, x_n}{M, \dots, M}$ $M, \dots, M$

FIGURE 17. Meta-layer of the SL system

are given in Figure 17.

### Meta-expressions

Meta-expressions extend expressions with meta-variables. In our implementation, meta-variables are the same as the variables in intermediate expressions. A set of meta-variables are used to distinguish them. In other words, a meta-expression is represented as a pair  $(S, M)$ , where  $S$  is a set of meta-variables and  $M$  is an expression. If a free variable of the expression is in the set, it is considered a meta-variable. Otherwise, it is a variable in the intermediate expression.

### Logic-expressions

Logic-expressions can have the following forms: primary meta-expressions, predicates on meta-expressions, high-level constraints on meta-expressions, and logic operations on logic-expressions. The predicates considered in this thesis are equal, less

then, one-step reduction, and multi-step reduction. The high-level constraints are special predicates used to check whether a meta-expression is a value and whether it can be matched by a context or abstract pattern. The logic operations include the regular boolean operations and, or, and not.

### Meta-theorems

Meta-theorems are represented by a tuple with a name, the set of meta-variables, a list of logic-expressions as premises, and a list of logic-expressions as conclusions.

In the SL language, we introduce a new phrase for subject reduction:

---

```
subred [x, gamma] x:Typeof(x, gamma) and x;;
```

---

It can be desugared to the following meta-theorem:

---

```
theorem subred: for all x, gamma,
if x ==>> y, Typeof(x, gamma) = t1 then Typeof(y, gamma) = t1
;;
```

---

The meta-theorem has two meta-variables  $x$  and  $\text{gamma}$ . It also has two premises and one conclusion.

## Meta-layer utilities

To automatically prove meta-theorems, some operations on the constructs of the meta-layer are needed. This section discusses the utilities of the meta-layer.

### 1. Meta-expression unification

Meta-expression unification is to find meta-variable instantiations on two meta-expressions so that the meta-expressions can be identified if the instantiations apply to the meta-expressions respectively.

The algorithm of unifying meta-expressions is given in Figure 18.  $\mathcal{U}$  is the unification function. The function takes a pair of meta-expressions,  $(m_1, m_2)$ , as arguments, and it returns a triple,  $(env_1, env_2, m_3)$ , where  $env_1$  and  $env_2$  are the bindings for the meta-expressions  $m_1$  and  $m_2$  respectively, and  $m_3$  is the resulting meta-expression.

An auxiliary function *merge\_env* is used in the algorithm, which merges two instantiation environments into one. If the same meta-variable instantiated in both environments, the corresponding expressions need to be unified and the resulting environment from the unification will also be merged into the final environment. The behavior of *merge\_env* is given in Figure 19.

For example, the meta-expressions  $m_1 : \text{App}(x, \sigma_1)$  and  $m_2 : \text{App}(\sigma_2, y)$  can be unified into  $m_3 : \text{App}(x, y)$  with instantiations  $(\sigma_1 : y)$  for  $m_1$  and  $(\sigma_2 : x)$  for  $m_2$ . The unification for meta-expressions  $\text{App}(x, \sigma_1)$  and  $\text{Lam}(\sigma_2, y)$  will fail, so is the unification of  $\text{App}(x, \sigma_1)$  and  $\text{App}(y, \sigma_2)$ .

The meta-expressions are first-order expressions. As shown in Chapter I, a mechanism is needed to equate lambda-expressions differing only in the names of bound

$$\mathcal{U}(m_1, m_2) = (env_1, env_2, m_3)$$

```

    case( $\sigma_1, \sigma_2$ ) : ( $\{(\sigma_1, \sigma_3)\}, \{(\sigma_2, \sigma_3)\}, \sigma_3$ )
    case( $\sigma_1, m_2$ )   : ( $\{(\sigma_1, m_2)\}, \{\}, m_2$ )
    case( $m_1, \sigma_2$ ) : ( $\{\}, \{(\sigma_2, m_1)\}, m_1$ )
    case( $x_1, x_1$ )     : ( $\{\}, \{\}, x_1$ )
    case( $c_0, c_0$ )     : ( $\{\}, \{\}, c_0$ )
    case( $c_1(s_1), c_1(s_2)$ ) : let ( $env_1, env_2, s_3$ ) =  $\mathcal{U}(s_1, s_2)$  in
                               ( $env_1, env_2, c_1(s_3)$ )
    case( $(s_{11}, s_{12}), (s_{21}, s_{22})$ ) : let ( $env_{11}, env_{21}, s_{31}$ ) =  $\mathcal{U}(s_{11}, s_{21})$  in
                                           let ( $env_{12}, env_{22}, s_{32}$ ) =  $\mathcal{U}(s_{12}, s_{22})$  in
                                           let  $env_1$  =  $merge\_env(env_{11}, env_{12})$  in
                                           let  $env_2$  =  $merge\_env(env_{21}, env_{22})$  in
                                           ( $env_1, env_2, (s_{31}, s_{32})$ )
    case( $\lambda x_1.s_1, \lambda x_2.s_2$ ) : let  $x$  =  $new\_variable$  in
                                           let  $s'_1$  =  $subst(s_1, x_1, x)$  in
                                           let  $s'_2$  =  $subst(s_2, x_2, x)$  in
                                            $\mathcal{U}(s'_1, s'_2)$ 
    case( $s_{11} s_{12}, s_{21} s_{22}$ ) : let ( $env_{11}, env_{21}, s_{31}$ ) =  $\mathcal{U}(s_{11}, s_{21})$  in
                                           let ( $env_{12}, env_{22}, s_{32}$ ) =  $\mathcal{U}(s_{12}, s_{22})$  in
                                           let  $env_1$  =  $merge\_env(env_{11}, env_{12})$  in
                                           let  $env_2$  =  $merge\_env(env_{21}, env_{22})$  in
                                           ( $env_1, env_2, s_{31} s_{32}$ )
    case( $s_{11}[s_{12}], s_{21}[s_{22}]$ ) : let ( $env_{11}, env_{21}, s_{31}$ ) =  $\mathcal{U}(s_{11}, s_{21})$  in
                                           let ( $env_{12}, env_{22}, s_{32}$ ) =  $\mathcal{U}(s_{12}, s_{22})$  in
                                           let  $env_1$  =  $merge\_env(env_{11}, env_{12})$  in
                                           let  $env_2$  =  $merge\_env(env_{21}, env_{22})$  in
                                           ( $env_1, env_2, s_{31}[s_{32}]$ )
    others : fail

```

FIGURE 18. Unification of meta-expressions

$$\text{merge\_env}(\text{env}_1, \text{env}_2) = \text{env}_3$$

$$\text{case}(\{\}, \text{env}_2) : \text{env}_2$$

$$\text{case}((\sigma_1 : m_1) :: \text{env}_1, \text{env}_2) : \text{let } \text{env}_3 = \text{merge\_env}(\text{env}_1, \text{env}_2) \text{ in}$$

$$\text{merge\_env}(\{(\sigma_1, m_1)\}, \text{env}_3)$$

$$\text{when } (\sigma_1, m_2) \notin \text{env}_2 \text{ and } \text{env}_1 \neq \{\}$$

$$\text{case}((\sigma_1 : m_1) :: \text{env}_1, \text{env}_2) : \text{let } \text{env}_3 = \text{merge\_env}(\text{env}_1, \text{env}_2) \text{ in}$$

$$(\sigma_1, m_1) :: \text{env}_3$$

$$\text{when } (\sigma_1, m_2) \notin \text{env}_2 \text{ and } \text{env}_1 = \{\}$$

$$\text{case}((\sigma_1 : m_1) :: \text{env}_1, (\sigma_1, m_2) :: \text{env}_2) : \text{let } \text{env}_3 = \text{merge\_env}(\text{env}_1, \text{env}_2) \text{ in}$$

$$\text{let } (\text{env}'_1, \text{env}'_2, m_3) = \mathcal{U}(m_1, m_2) \text{ in}$$

$$\text{let } \text{env}'_3 = \text{merge\_env}(\text{env}'_1, \text{env}'_2) \text{ in}$$

$$\text{let } \text{env}''_3 = \text{merge\_env}(\text{env}_3, \text{env}'_3) \text{ in}$$

$$\text{merge\_env}(\{(\sigma_1, m_3)\}, \text{env}''_3)$$

FIGURE 19. Merging two environments

variables ( $\alpha$  equivalence). This is achieved by using meta-expression unification.  $\alpha$ -equivalence is easily determined by unifying expressions while considering the free variable and bound variable information.

## 2. Meta-pattern matching:

Meta-pattern matching is matching primary meta-expressions with patterns which are defined as in Chapter II. Meta-pattern matching does not build automata, instead it creates bindings for both patterns and meta-expressions.

Figure 20 shows the process of meta-pattern matching, which is described by a function  $\mathcal{P}$ . The function  $\mathcal{P}$  takes two arguments,  $(p, m)$ , where  $p$  is a pattern and  $m$  is a meta-expression. The function returns a triple,  $(env1, env2, ll)$ , where  $env1$  is the environment that records the bindings of variables in the pattern  $p$ ,  $env2$  is the environment that records the bindings of meta-variables in the meta-expression, and  $ll$  is the logic expressions created in the meta-pattern matching.

The function  $\mathcal{P}$  focuses on the case when the meta-expression argument is a meta-variable. The other cases can be reduced to the meta-variable case by adding a binding to the meta-variable afterwards. For example,  $\mathcal{P}(p, m)$  can be reduced to  $\mathcal{P}(p, \sigma)$  whose return triple is used for the return value of  $\mathcal{P}(p, m)$  except that the environment of meta-variable  $\sigma$  is added with the binding  $\sigma = m$ .

## 3. Meta-evaluator

Meta-evaluation is the evaluation of meta-expressions. It simplifies the meta-expressions by using general evaluation rules. The rules are presented in Figure 21. The function  $\mathcal{E}$  is the evaluation function, which takes an environment and a meta-expression as arguments, and returns a modified environment and a



$$\mathcal{P}(p, m) = (env_1, env_2, ll)$$

```

case(., σ) : ( {}, {}, {} )
case(x, σ) : ( {(x, σ)}, {}, {} )
case(const, σ) : ( {}, {(σ, const)}, {} )
case(c0, σ) : ( {}, {(σ, c0)}, {} )
case(c1p1, σ) : let (env1, env2, ll) = P(p1, σ1) in
                  (env1, merge_env(env2, {(σ, c1 σ1)}), ll)
case(p1asx, σ) : let (env1, env2, ll) = P(p1, σ1) in
                  (merge_env(env1, {(x, σ)}), env2, ll)
case((p1 : type), σ) : P(p1, σ)
case((p1, p2), σ) : let (env11, env21, ll1) = P(p1, σ1) in
                  let (env12, env22, ll2) = P(p2, σ2) in
                  (merge_env(env11, env12),
                   merge_env(env21, env22, {(σ, (σ1, σ2))}),
                   ll1 ∪ ll2)
case((p1 : dyn), σ) : let (env1, env2, ll) = P(p1, σ) in
                  (env1, env2, ll ∪ {mpat(dyn, σ)})
case((p1 : apat)p2, σ) : let (env11, env21, ll1) = P(p1, σ1) in
                  let (env12, env22, ll2) = P(p2, σ2) in
                  (merge_env(env11, env12),
                   merge_env(env21, env22, {(σ, σ1 σ2)}),
                   ll1 ∪ ll2 ∪ {mpat(apat, (σ1, σ2))})

```

---

```

case(□, σ) : ( {}, {(σ1, λx.x)}, {σ = σ1 σ2} )

```

```

case(p, m) : let (env1, env2, ll) = P(p, σ) in
             (env1, env2, ll ∪ {σ = m})

```

FIGURE 20. Meta-pattern matching

simplified meta-expression.

### An Example of Automatically Proving Subject Reduction by Induction

This section shows the process of automatically proving subject by using the example in Figure 16.

The proof is by induction. At first, induction is on the number of reduction steps of  $x \Rightarrow y$ . This induction is trivial and we reduce the theorem to:

---

```
theorem subred: for all x, gamma,
if x ==> y, Typeof(x, gamma) = t1
then Typeof(y, gamma) = t1;;
```

---

The proof of subject reduction needs two lemmas. One is a substitution lemma and the other is a context lemma.

The substitution lemma is stated as:

#### Lemma 7

If  $\text{Typeof}(m1, \text{gamma}) = t1$  and  $\text{Typeof}(m, (x:t1)::\text{gamma}) = t2$  then  $\text{Typeof}(m[m1/x], \text{gamma}) = t2$ .

The context lemma is stated as:

#### Lemma 8

If  $\text{Typeof}(m, \text{gamma}) = \text{Typeof}(n, \text{gamma})$  then  $\text{Typeof}(H[m], \text{gamma}) = \text{Typeof}(H[n], \text{gamma})$  for any context  $H$ .

$$\mathcal{E}(env, m) = (env', m')$$

```

case(env,  $\sigma$ ) : (env,  $\sigma$ )  when ( $\sigma, m'$ )  $\notin$  env
case(env,  $\sigma$ ) : (env,  $m'$ )  when ( $\sigma, m'$ )  $\in$  env
case(env,  $x$ )   : (env,  $x$ )
case(env, const) : (env, const)
case(env,  $c_0$ )  : (env,  $c_0$ )
case(env,  $c_1 m_1$ ) : let (env1,  $m'_1$ ) =  $\mathcal{E}(env, m_1)$  in
                    (env1,  $c_1 m'_1$ )
case(env, ( $m_1 : type$ )) : let (env1,  $m'_1$ ) =  $\mathcal{E}(env, m_1)$  in
                    (env1, ( $m'_1 : type$ ))
case(env, ( $m_1, m_2$ )) : let (env1,  $m'_1$ ) =  $\mathcal{E}(env, m_1)$  in
                    let (env2,  $m'_2$ ) =  $\mathcal{E}(env, m_2)$  in
                    (merge_env(env1, env2), ( $m'_1, m'_2$ ))
case(env, op( $m_1, m_2$ )) : let (env1,  $m'_1$ ) =  $\mathcal{E}(env, m_1)$  in
                    let (env2,  $m'_2$ ) =  $\mathcal{E}(env, m_2)$  in
                    let  $m' = \text{built-in}(op, m'_1, m'_2)$  in
                    (merge_env(env1, env2),  $m'$ )
case(env,  $\lambda x. m_1$ ) : let (env1,  $m'_1$ ) =  $\mathcal{E}(env, m_1)$  in
                    (env1,  $\lambda x. m'_1$ )
case(env,  $m_1 m_2$ ) : let (env1,  $m'_1$ ) =  $\mathcal{E}(env, m_1)$  in
                    let (env2,  $m'_2$ ) =  $\mathcal{E}(env, m_2)$  in
                    let  $m' = m'_1 m'_2$  in
                    (merge_env(env1, env2),  $m'$ )
case(env,  $m_1[m_2]$ ) : let (env1,  $m'_1$ ) =  $\mathcal{E}(env, m_1)$  in
                    let (env2,  $m'_2$ ) =  $\mathcal{E}(env, m_2)$  in
                    let  $m' = m'_1[m'_2]$  in
                    (merge_env(env1, env2),  $m'$ )

```

FIGURE 21. Meta-expression evaluation

We apply induction on the cases of  $x \implies y$  by matching the axiom and inference rule. It may match either the beta axiom or the inference rule. Meta-patternmatching is performed so that the meta variable  $x$  will be instantiated in different cases.

case1: match beta axiom,  $x$  is bound to  $\text{App}(\text{Abs}(s,x1),x2)$ . Two new meta-variables are introduced but they are only part of the original meta-variables.  $y$  is bound to  $x2[x1/s]$  according to the right-hand side of the beta axiom. The sub-theorem becomes:

---

```
for all s, x1,x2, Gamma:
if Typeof(App(Abs(s,x1),x2), Gamma) = t1
then Typeof(sp\_subst(x1,s,x2), Gamma) = t1;;
```

---

Using the eval rule of Typeof on App, the theorem to be proven becomes:

---

```
for all s, x1, x2, Gamma:
if Typeof(Abs(s,x1), Gamma) = TArrow(t11,t1) and
   Typeof(x2, Gamma) = t11
then Typeof(sp\_subst(x1,s,x2), Gamma) = t1
```

---

Using the eval rule of Typeof on Abs, the theorem to be proven becomes:

---

```
for all s,x1,x2,Gamma:
```

```

if Typeof(x1, (s:t11)::Gamma) = t1 and
  Typeof(x2, Gamma) = t11
then Typeof(x1[x2/s], Gamma) = t1

```

---

Then we apply the substitution lemma, and the case is proved.

case2: match inference rule,  $x$  is bound to  $E[x']$ . Both  $E$  and  $x'$  are new meta-variables.  $y$  is bound to  $E[y']$  where  $x' \implies y'$ . The sub-theorem becomes:

---

```

for all x', E, Gamma,
if x' ==> y' and Typeof(E[x'], Gamma) = t1
then Typeof(E[y'], Gamma) = t1

```

---

Introducing a temp variable  $t1' = \text{Typeof}(x', \text{Gamma})$

---

```

for all x', E, Gamma,
if x' ==> y', Typeof(x', Gamma) = t1', and
  Typeof(E[x'], Gamma) = t1
then Typeof(E[y'], Gamma) = t1

```

---

Using the hypothesis

---

for all  $x'$ ,  $\Gamma$ : if  $x' \Rightarrow y'$ ,  $\text{Typeof}(x', \Gamma) = t1'$   
 then  $\text{Typeof}(y', \Gamma) = t1'$

---

The theorem to be proven becomes:

---

for all  $x'$ ,  $E$ ,  $\Gamma$ :  
 if  $x' \Rightarrow y'$ ,  $\text{Typeof}(x', \Gamma) = t1'$ ,  
 $\text{Typeof}(y', \Gamma) = t1'$  and  
 $\text{Typeof}(E[x'], \Gamma) = t1$   
 then  $\text{Typeof}(E[y'], \Gamma) = t1$

---

By using the context lemma, the case is proved.

The subject reduction lemma is then since both cases are proved. The SL system prints:

---

Theorem subred has been proved.

---

#### Automatically Proving Subject Reduction by Induction

Generally, the inductive proof algorithm is given as follows: Given a theorem,

1. We check whether it is an instance of the theorems already known to be proved, a hypothesis, or already rejected. This is done by searching the tables containing all the theorems with known results.
2. If it is not known, check whether the depth of proofs exceeds the limit. If so, let user to choose whether to continue or abort.
3. If the user chooses to continue, or the depth has not exceeded the limit, we put the theorem in the hypothesis table, and do the following analysis on the most important premise.
  - If the premise is a rewriting step, do case analysis on the axioms or inference rules. For each matched rules, we simplify the theorem, hence we obtain a list of sub-theorems to prove. Each sub-theorem in the list has to be true in order to make the theorem true. The process for each sub-theorem follows the same algorithm, only with the proof depth incremented.

If the list is empty, the theorem is vacuously proven.
  - If the premise is a sig-function pattern, do case analysis on the rules or the alternatives of the sig-function. For each matched one, we simplify the theorem, hence we obtain a list of sub-theorems. Each sub-theorem in the list has to be tried in order to make the theorem true. The process for each sub-theorems follows the same algorithm, only with the proof depth incremented.

If the list is empty, the theorem is proved vacuously.
  - If the premise is a logic premise, try to simplify the logic premise. If the theorem can be simplified, use the algorithm to process the simplified theorems.

The depth is not increased here.

- If the premise is a logic constraint, replace the logic constraint with simpler premises by pattern matching the logic constraint. For example, if the constraint is that an expression  $M$  can be decomposed into a context  $C$  and an inner expression  $N$ , it uses the rules for contexts and instantiates the expression  $M$  to  $C[N]$  with all the possible contexts  $C$ , producing all the cases needed to be prove the original theorem.
- If the premise is a logic operation, like logical OR, AND, and NOT, it simplifies the theorem with the logic operation rule.

For logical OR, it derives two sub-theorems. Each theorem is a copy of the original theorem except that the premise under consideration is replaced with one of operands of logical OR. If one of the sub-theorems can be proven, then the original theorem is proved.

For logical AND, it derives two sub-theorems. Each theorem is a copy of the original theorem except that the premise under consideration is replaced with one of operands of logical AND. If both of the sub-theorems can be proven, then the original one is proved.

For logic NOT, it derives a sub-theorems which is a copy of the original theorem except that the premise under consideration is replaced with the operand of the logical NOT. If the sub-theorem is proved to be false, then the original theorem is proved.

Each of the cases above increase the depth when proving sub-theorems.

4. The user may choose to abort, stop the current proving path, and track back to try



other paths.

5. All the sub-theorem proving is done by applying the same algorithm. When the original theorem is proved or disproved, remove the hypothesis for the original theorem.

### Summary and Discussion

Subject reduction property is essential for soundness of type system of programming language. We describe automatically checking such property in the SL system in this chapter.

The SL system is enhanced with a meta-layer so that subject reduction lemma is represented as a meta-theorem, which consists of a list of logic expressions as premises and a list of logic expressions as conclusions. The meta-layer also contains the necessary utilities for automatic proving, such as meta-unification, meta-patternmatching, meta-evaluation, and meta-theorem management.

The automatic proving is inductive. Induction can be made on the number of reduction steps, the rewriting rules, the type checking rules, and the structure of data-type definition of the object terms. Each induction step transforms a meta-theorem into simpler meta-theorems. It either instantiates meta-variables with expressions, or evaluates meta-expressions in the meta-theorem. The base case is reached in one of the following conditions:

- The meta-theorem can be easily validated as logic tautology
- The meta-theorem is an instance of proved theorem
- The meta-theorem is an instance of active hypothesis.

The meta-framework is currently specialized for proving subject reduction lemma. However, it is extensible for automatically proving other utilites.

## CHAPTER VI

### CONCLUSION AND FUTURE WORK

The SL system is a semantic toolkit for manipulating syntactic theories. Its basic function is to read a syntactic theory for a language and to produce an interpreter for that language. The syntactic theory is represented in the domain-specific SL language and the generated interpreter is a program in CAML. The SL system is also able to automatically prove or to check some properties about the specified syntactic theories. The properties that SL currently handles are decomposition and subject reduction.

The SL system is available from:

<http://www.cs.uoregon.edu/~ariola/SL/>.

In this chapter, we summarize the contributions and applications of the SL system. We then discuss directions for future work.

#### Contributions and Applications of the SL System

The contributions of this work include the following:

- We designed a domain-specific language, SL, for writing syntactic theories.

Many notions for syntactic theories like dynamic expressions, contexts, abstract patterns, axioms, and inference rules are among the special features in the SL language. Therefore, representations of syntactic theories in the SL language are more straightforward and readable.

- We introduced new forms of patterns.

The new forms of patterns include dynamic constraint patterns, context constraint patterns, and abstract patterns. They provide an expressive and flexible means for abstracting a large variety of terms. Abstract patterns can be extended to other languages with patterns.

- We implemented a compiler for the SL language.

the SL compiler generates a parser and pretty-printer for any specified data-type. The compilation also includes an extension of the pattern matching algorithm to handle the new patterns.

- We explored two methods for automatically checking or proving properties of syntactic theories.

For unique decomposition, our method is to map the property of syntactic theories to a property of tree automata. For subject reduction, our method is to use inductive reasoning in a meta-layer. The latter method can be extended to check or prove other properties.

We have shown examples of the use of the SL system in this thesis. In general, the applications of the SL system could be the following:

- Language study:

People studying programming languages can use the SL system to understand the language semantics and the evaluation of input terms. Given semantic rules, the interpreters generated by the SL system can show how a term is evaluated. At each step of reduction, the interpreter shows which rules are used and what the position of the redex is.

Another student in our group had developed a GUI based on the SL system. The GUI makes it more convenient to use the SL system and makes it clearer to view the evaluation of terms. Multiple windows are used in the GUI. There are windows for editing semantic rules and input terms. When the evaluation button or the step-by-step reduction button is pressed, a new window will pop-up with the reduction sequence. The evaluation contexts and redexes are shown with different colors for each step. When the semantic rules are changed, one can instantly see the change in evaluation.

– Prototyping compilers/interpreters:

Because the interpreters are generated automatically, the SL system is a good candidate for prototyping compilers/interpreters. One only needs to represent the abstract syntax and semantic rules in the SL language. If concrete syntax needs to be addressed, a parser to parse a program to an abstract term is also needed.

– Code generation:

The SL system is not only for generating interpreters. If a process can be abstracted into rules, the SL system can generate code for such a process. For example, the transformation between two data-structures can normally be specified with rules, hence code for a such transformation can be easily obtained by using the SL system.

### Future work

The SL system is still in an experimental system. Future work can address the following areas:

– Specification Languages:

More features can be added to the SL language. Some important notions such as normal forms and term graphs can be added. In addition, SL specifications could be made modular, so that it would be easy to import or export data-types and rules. Additional built-in constructs, such as sets, stacks, and hash tables could also be considered. These constructs could make SL more powerful for specifying complicated data-types.

– More efficient compilation:

The pattern matching algorithm described in Chapter II is efficient for one-step reduction. But after a reduction occurs, matching for the next step has to begin from scratch. It would be more efficient if the intermediate results of pattern matching from the previous step could be saved and used for the next step. The implementation used for automatically proving decomposition and subject reduction also need efficiency improvement.

– Code optimization:

There is plenty of room for optimizing the generated code, which is not the focus of the current SL system. For example, many functions corresponding to some states can be inlined. This would make the generated code more readable and improve the performance of the generated executable.

– Property Checking:

Other properties could be considered. The meta-layer framework used for proving subject reduction needs to be tuned to be more general. Other methods could also be explored for performing the automatic checking/proving of properties.

## APPENDIX

## THE SL LANGUAGE

We describe a BNF of the SL language in this appendix, where `program` is the start non-terminal. The following are the notations used for the BNF:

- Texts enclosed between “'”s are terminals.
- Texts enclosed between “<” and “>” are non-terminals.
- Texts enclosed between “/\*” and “\*/” are comments.
- “{ part }\*” means zero or multiple concatenence of the part.
- “{ part }?” means zero or one occurence of part.

The BNF of the SL language:

---

```

program = 'SIGNATURE:' <signature_part>
        'SPECIFICATION:' <specification_part>

signature_part = {<sigtype_decls>}* <startfrom_phrase>
                {<sigtype_decls>}*

sigtype_decls = 'type' <sigtype_decl1> {and <sigtype_decl1>}* ';;'

sigtype_decl1 =
    (''ident, ..., ''ident) ident '==' <sigtype_expr>

```

```
| {''ident, ..., ''ident) ident '='
    <constr_def1> {'|' <constr_def1>}*
```

```
constr_def1 = ident 'of' <sigtype_expr>
```

```
sigtype_expr = ''ident
    | <sigtype_expr>* ident
    | <sigtype_expr> { '*' <sigtype_expr> }*
    | '(' <sigtype_expr> ')'
```

```
startfrom_phrase = 'startfrom <sigtype_expr> ';;'
```

```
specification_part =
    <type_decl_phrase> /* see caml-light */
    | <exception_phrase> /* see caml-light */
    | <letdef_phrase> /* see caml-light */
    | <expr_phrase> /* see caml-light */
    | <direction_phrase> /* see caml-light */
    | <dynamic_decl_phrase>
    | <context_decl_phrase>
    | <axiom_phrase>
    | <inference_phrase>
    | <strategy_phrase>
```

```
dynamic_phrase = 'dynamic' <dynamic_decl1>
    ('and' <dynamic_decl1>)* ';;'
```



```
dynamic_decl1 = ident '=' <dyn_expr> { '|' <dyn_expr> }*
```

```
dyn_expr = <constant>
```

```

| '_'
| ident /* type name */
| <constructor> { <dyn_expr> }* /* including list_cons */
| <dyn_expr> { ',' <dyn_expr> }*
| '(' <dyn_expr> ')'
| '[' { <dyn_expr> { ';' <dyn_expr> }* }? ']'

```

```
context_phrase = 'context' <context_decl1>
```

```
{ 'and' <context_decl1> }* ';;'
```

```
context_decl1 = ident '=' <ctxt_expr> { '|' <ctxt_expr> }*
```

```
ctxt_expr = <constant>
```

```

| '_'
| ident /* type name */
| <constructor> { <dyn_expr> }* /* including list_cons */
| <ctxt_expr> { ',' <ctxt_expr> }*
| '(' <ctxt_expr> ')'
| '[' { <ctxt_expr> { ';' <ctxt_expr> }* }? ']'
| 'BOX'
| context_name <ctxt_expr> /* context apply */

```

```
axiom_phrase = 'axiom' {ident}? {'with' 'startfrom'}? ':'
              <axiom_item> { '|' <axiom_item> }* ';;'
```

```
axiom_item = <lhs> <condition> '==>' <rhs>
```

```
lhs = ...patterns allowed in caml-light, except record pattern...
```

```
| '(' <lhs> ':' <dyn_name> ')'
    /* lhs should be wild or var */
| '(' <lhs> ':' <ctxt_name> ')'
    /* lhs should be wild or var */
| '(' <lhs> ':' 'dynamic' <dyn_expr> ')'
    /* lhs should be wild or var */
| '(' <lhs> ':' 'context' <ctxt_expr> ')'
    /* lhs should be wild or var */
| <lhs> <lhs>
    /* the left <lhs> should be a context constraint */
```

```
condition = /* epsilon */
           | 'when' <expression>
```

```
rhs = <expression>
```

```
inference_phrase =
  'inference' {ident}? {'with' 'startfrom'}? ':'
  <inference_item> { '|' <inference_item> }* ';;'
```

```
inference_item =  
    <rhs> '==>' <lhs>  
    '-----' /* any number greater than 4 */  
    <lhs> <condition> '==>' <rhs>  
  
strategy_phrase = 'strategy' {<strategy_spec1>}+  
  
strategy_spec1 = leftmost | rightmost  
                | topdown | bottomup  
                | innermost | outermost
```

---

## BIBLIOGRAPHY

- [AB97] Zena M. Ariola and Stefan Blom. Cyclic lambda calculi. In the *International Symposium on Theoretical Aspects of Computer Software (TACS)*, 1997.
- [AF97] Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *J. Functional Programming*, 7(3), 1997.
- [AFM<sup>+</sup>95] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 233–246. ACM Press, New York, 1995.
- [ASU85] A. Aho, R. Sethi, and J. Ullman. *Compilers—Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1985.
- [Bar84] Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science Publishers B.V., Amsterdam, revised edition, 1984.
- [BBKM93] David A. Basin, Alan Bundy, Ina Kraan, and Seán Matthews. A framework for program development based on schematic proof. In *Proceedings of the 7th International Workshop on Software Specification and Design*, pages 162–171. IEEE Computer Society Press, 1993.
- [BCD<sup>+</sup>88] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Proceedings of SIGSOFT'88, Third Annual Symposium on Software Development Environments (SDE3)*, Boston, USA, 1988.
- [BH91] Rod Burstall and Furio Honsell. Operational semantics in a natural deduction setting. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 185–214. Cambridge University Press, 1991.
- [C<sup>+</sup>86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
- [Cam] The CAML Language Homepage <http://caml.inria.fr/>.
- [CDG<sup>+</sup>99] Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemart, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree Automata Techniques and Applications. Book draft available at <http://www.grappa.univ-lille3.fr/tata>, April 1999.

- [Coq] The Coq Project Homepage <http://pauillac.inria.fr/coq>.
- [DFH<sup>+</sup>93] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Technical Report Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.
- [DPS97] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In R. Hindley, editor, *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications*, pages 147–163, Nancy, France, April 1997. Springer-Verlag, Berlin.
- [ELA] The ELAN System Homepage <http://www.loria.fr/ELAN>.
- [FB97] Manuel Fähndrich and John Boyland. Statically checkable pattern abstractions. In the *ACM SIGPLAN International Conference on Functional Programming*, pages 75–84. ACM Press, New York, 1997.
- [FLS99] John Fiskio-Lasseter and Amr Sabry. Putting operational techniques to the test: A syntactic theory for behavioral Verilog. *Electronic Notes in Theoretical Computer Science*, 26:32–49, 1999. Third International Workshop on Higher Order Operational Techniques in Semantics.
- [Gar92] Philippa Gardner. *Representing Logics in Type Theory*. PhD thesis, University of Edinburgh, July 1992. Available as Technical Report CST-93-92.
- [GS84] Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.
- [HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
- [HM89] John Hannan and Dale Miller. A meta-logic for functional programming. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming*, chapter 24, pages 453–476. MIT Press, 1989.
- [How80] W. A. Howard. The Formulae-As-Types Notion Of Construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, Inc., New York, N.Y., 1980.
- [HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Computer Science. Addison-Wesley, 1979.

- [Isa] The Isabelle System Homepage <http://www.cl.cam.ac.uk/Research/HVG/Isabelle>.
- [Kah87] G. Kahn. Natural semantics. In *Proc. STACS*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, Berlin, 1987.
- [Ler90] X. Leroy. The Zinc experiment: An economical implementation of the ML language. Technical Report 117, INRIA, 1990.
- [IPr] The  $\lambda$ Prolog System Homepage <http://www.cse.psu.edu/~dale/lProlog>.
- [LS97] John Launchbury and Amr Sabry. Monadic state: Axiomatization and type safety. In the *ACM SIGPLAN International Conference on Functional Programming*, pages 227–238. ACM Press, New York, 1997.
- [Mar94] Luc Maranget. Two techniques for compiling lazy pattern matching. Research report 2385, INRIA, 1994.
- [Mas99] Ian A. Mason. Computing with contexts. *HIGHER ORDER SYMBOLIC COMPUTATION*, 12(2):171–201, September 1999.
- [Mil89] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Proceedings of the International Workshop on Extensions of Logic Programming*, pages 253–281, Tübingen, Germany, 1989. Springer-Verlag LNAI 475.
- [MK98] Pierre-Etienne Moreau and H el ene Kirchner. A compiler for rewrite programs in associative-commutative theories. In “*Principles of Declarative Programming*”, number 1490 in *Lecture Notes in Computer Science*, pages 230–249. Springer-Verlag, September 1998. Report LORIA 98-R-226.
- [MN94] Lena Magnusson and Bengt Nordstr om. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 213–237. Springer-Verlag LNCS 806, 1994.
- [NM88] Gopalan Nadathur and Dale Miller. An overview of  $\lambda$ Prolog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.
- [Nup] The Nuprl Project Homepage  
<http://simon.cs.cornell.edu/Info/Projects/Nuprl/nuprl.html>.
- [Pau90] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

- [PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In the *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208, June 1988.
- [Plo75] Gordon D. Plotkin. Call-by-name, call-by-value, and the  $\lambda$ -calculus. *Theoret. Comput. Sci.*, 1:125–159, 1975.
- [SS99] Miley Semmelroth and Amr Sabry. Monadic encapsulation in ML. In the *ACM SIGPLAN International Conference on Functional Programming*, pages 8–17. ACM Press, New York, 1999.
- [str] The Stratego System Homepage  
<http://www.stratego-language.org/Stratego/WebHome>.
- [Twe] The Twelf System Homepage <http://www.cs.cmu.edu/~twelf>.
- [Vis01] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.
- [WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, November 1994.