Contents

Chapter 1

Introduction

Term rewriting and *unification* are two areas of computer science, which study the paradigms of computational procedures. *Matching* as a special case of unification is the most important technique for term rewriting. Rewriting systems reflect the idea of simplifying terms, which has been around as long as algebra. As a form of computer program, many modern programs for symbolic manipulation use rewrite rules for simplification. As a formalism, rewriting systems have the full power of Turing machines and may be thought of as nondeterministic algorithms over terms, rather than strings. The mathematical investigation of equation solving is a subject as old as mathematics itself, and right from the beginning was very much at its heart. Unification carries this activity over into computer science, and can be seen as the "addition and multiplication" of many deduction systems, such as theorem provers and logic programming.

This thesis studies matching problems together with their applications in term rewriting. We present a framework for defining problems, develop matching algorithms, prove the properties of the algorithms, and show the applications of our approach. In particular, we consider the application in program transformation systems.

In this chapter, we first give a general introduction to unification and matching, and a brief discussion about term rewriting as a formal computational system and its applications, by which we hope to provide enough background knowledge and references. Then, we discuss our research motivation and give an overview of the thesis. Related work is decentralized in the individual chapters.

1.1 Matching and Unification

As matching is a special case of unification, often called one-side unification, we should begin the discussion with unification.

Since the publication of Herbrand's unification algorithm in 1930 and Robinson's formal unification algorithm for first-order terms in 1965, unification has received considerable attention. There is still no ultimate definition of unification. To present unification in a general way, we take the following definition: Unification shall solve equations between terms formed from a set $\mathcal{F}$ of function symbols and a set $\mathcal{V}$ of variables in an arbitrary $\mathcal{F}$-algebra $\mathcal{A}$. A *unification problem* is often presented as

$$s =^?_{\mathcal{A}} t.$$

A solution, or *unifier*, of the problem is a substitution σ such that $\sigma s =_{\mathcal{A}} \sigma t$. If t contains no variable, then the unification problem is called a *matching problem*. A *match* is a substitution σ such that $\sigma s =_{\mathcal{A}} t$.

According to this definition, the research in unification concerns the following subjects.

Syntactic unification

Syntactic unification is the problem of solving equations in free algebra. A syntactic unification problem is written as $s =^? t$. A unifier of the problem is a substitution σ such that σs and σt are syntactically identical. Two important properties of syntactic unification are:

- decidable — it is decidable whether a problem is solvable or not.
- unitary — if a problem is solvable, then a unique minimal unifier, called *the most general unifier*, always exists.

Since Robinson, many people sought for more efficient unification algorithms to solve syntactic unification problems. One of the most influencing results is due to Martelli and Montanari [52], where they pointed out that unification should be seen as a process of solving equations. Their algorithm incorporates the detection of cycles along the way (rather than at the end) and performs well in practice. The matching algorithms in this thesis are all constructed from this point of view.

Semantic unification

Generally, *semantic unification*, also called *$\mathcal{E}$-unification*, is the problem of solving equations in a quotient algebra $\mathcal{A}/\mathcal{E}$, for a priori given set $\mathcal{E}$ of equational axioms. An $\mathcal{E}$-unification problem is often written as $s =^?_{\mathcal{E}} t$. An $\mathcal{E}$-unifier of the problem is a substitution such that σs and σt belong to the same equivalence class of the quotient algebra.

When it comes to $\mathcal{E}$-unification, the nice properties of syntactic unification are lost. It is not always decidable whether an $\mathcal{E}$-unification problem has a unifier or not. Even if a problem is solvable, the most general unifier usually does not exist. Decidable $\mathcal{E}$-unification can be divided into the following three classes (let $CMGU_{\mathcal{E}}(\Gamma)$ denote a *complete set of most general $\mathcal{E}$-unifiers* for the problem Γ):

- unitary: $CMGU_{\mathcal{E}}(\Gamma)$ contains at most one unifier for every Γ;
- finitary: $CMGU_{\mathcal{E}}(\Gamma)$ contains only finitely many unifiers for every Γ;
- infinitary: $CMGU_{\mathcal{E}}(\Gamma)$ contains infinitely many unifiers for some Γ.

A useful and important class of equational axioms is sets of convergent rewrite rules. An effective equation solving approach, called *narrowing*, has been discovered and proved to be complete to solve this class of $\mathcal{E}$-unification problems, see [21, 41].

Higher-order unification

Higher-order unification is the problem of solving equations in typed λ-calculus, i.e., of computing higher-order substitutions σ for the free variables of the equation $s =^? t$, which makes s and t equal modulo the conversion rules of the calculus. This problem is, of course, related to semantic first-order unification.

Higher-order unification is the key to higher-order theorem proving and higher-order logic programming. It is also crucial for program synthesis and program transformation [29, 40]. The main results on higher-order unification are:

- Second-order matching is decidable and finitary, see Huet and Lang's paper [40].
- Second-order unification is undecidable, see [28] by Goldfarb.
- Higher-order pattern unification is decidable and unitary. This result is due to Miller [55], who discovered a subclass of λ-terms, called *higher-order patterns*, and proved that higher-order pattern unification has the nice properties of syntactic unification.

Recently, Synder and Gallier gave a general overview of the work in this field and showed how Huet's procedure can be described and proved using transformation rules [79]. This highlights similarities and differences between first- and higher-order unification by expressing the fundamental elementary unification steps in the same way.

Higher-order $\mathcal{E}$-unification

A new research area in unification is the so-called *higher-order $\mathcal{E}$-unification*, which solves higher-order unification problems in a quotient algebra for some higher-order equational axioms $\mathcal{E}$. There is only little work published. To the best of our knowledge, Prehofer's work [70] gives the first framework for solving higher-order equations by several higher-order narrowing approaches. In that work, it is proved that the first-order narrowing technique can be lifted to solve higher-order $\mathcal{E}$-unification problems without loss of its completeness.

Why do we study matching in this thesis, a special case of unification? Except for practical considerations (see the next section), there are also two important theoretical reasons: complexity and efficiency. Usually, matching problems are less complex than unification problems. For example, A-matching is NP-complete, but one does not know yet whether A-unification is NP-complete or NP-hard, where A presents the associativity axiom; in the higher-order case, second-order and third-order matching are decidable [40, 18], but second-order unification is undecidable [28]. In addition, matching algorithms are usually more efficient than unification algorithms. In unification, variables in both terms should be replaced to make them equal, but in matching, only variables in one term need to be considered.

The *extended matching* discussed in this thesis consists of

- semantic matching,
- contextual matching — matching with additional conditions, and
- higher-order $\mathcal{E}$-matching.

1.2 Matching and Term Rewriting Systems

Many problems that can be solved or just presented by term rewriting systems are in general natural. The theory of rewriting is in essence a theory of normal forms; to some extent it is an outgrowth of the study of Church's Lambda Calculus and Curry's Combinatory Logic. There are a lot of problems that can be *presented* as term rewriting systems, in which matching plays a crucial role. We only take some of them as examples.

Functional languages

Term rewriting is the operational principle of most functional languages like ML [33], where functions are defined through the connection between arguments and results. For example, using the following two equations we define the concatenation function ++ on lists in ML:

$$\begin{array}{ll} \texttt{fun} & nil ++ ys \; = \; ys \\ \mid & (x :: xs) ++ ys \; = \; x :: (xs ++ ys) \end{array}$$

The left-hand sides are usually called *patterns*. To evaluate functions, we need to do pattern-matching, which belongs to syntactic matching.

λ-calculus

For half a century, the λ-calculus, probably the most well-known term rewriting system, has been playing a crucial role in computer science, particularly in programming languages. In λ-calculus, α-, β-, and η-reduction decide the rewriting relation on λ-terms. The following is the β-reduction

$$(\lambda x.t)s \;\rightarrow t[s/x].$$

Note that the left-hand side of the rule contains the bound variable x. So, to rewrite terms with this rule, we need to solve higher-order matching problems.

Program transformation systems

In general, program transformation systems, such as CIP [6] and PROSPECTRA [37], can also be treated as rewriting systems, where terms are programs and rewriting relations are decided by transformation rules. To cope with complex program schemata, transformation rules are usually defined on complicated patterns unlike in functional programming.

The following is a program transformation rule to optimize an "IF-THEN-ELSE" statement, which lifts the common part of "THEN" and "ELSE" statements.

$$\text{IF } x_c \text{ THEN } (s ++ s_1) \text{ ELSE } (s ++ s_2) \Longrightarrow s ++ (\text{IF } x_c \text{ THEN } s_1 \text{ ELSE } s_2) \Leftarrow hd\ s_1 \;\neq\; hd\ s_2$$

The rule contains the concatenation function on its left-hand side, and has an additional condition $hd\ s_1 \neq hd\ s_2$. Using the rule to transform programs involves semantic matching problems.

The transformation from recursion to iteration is a familiar problem. A basic schema for a linear recursion is

$$f =_{def} \lambda x.\ \text{IF } B\ x \text{ THEN } f(H\ x) \oplus (K\ x) \text{ ELSE } T\ x,$$

which defines the function f in a linearly recursive way. Except that the bound variable x may express a list, this schema contains higher-order variables like B, H, K and T. To match a program with the schema is a higher-order matching problem.

These examples show that matching problems in program transformation systems have a certain complexity. This is an important reason leading us to choose program transformation as an application area.

1.3 Goals and Overview of this Thesis

The idea that has motivated this research is quite simple and practical: to have a powerful notion of matching to support program transformations. As a result, we obtain a general framework of matching, which does meet our original expectation and can be applied to many other rewriting-oriented areas as well. Our basic goals can be summarized as follows:

- to study a powerful notion of matching, which enhances the application of rewriting-oriented systems, particularly program transformation systems;
- to develop matching algorithms, which not only have nice theoretical properties (completeness, soundness, and termination); but are also practically useful: they enable easy and efficient implementations;
- to make use of existing theoretical results and techniques to support our research.

We first present some important new concepts introduced in this thesis; this may help to obtain an overview of the work.

- **Constructor-based specifications** of both first- and higher-order define terms and sets of equational axioms. An important property is that they can be treated as convergent term rewriting systems.
- **Structural narrowing** is a new narrowing technique introduced to solve the semantic matching problems; it is proved to be complete.
- **Matching combinators** are functions defined by constructor-based specifications. Semantic matching problems containing matching combinators can be efficiently solved by using structural narrowing.
- **Acyclic matching combinators** constitute a subclass of matching combinators. Semantic matching problems containing only acyclic matching combinators are finitary and decidable.
- **Extended patterns** extend the class of higher-order patterns (see [55]) and ensure the effective solvability of higher-order $\mathcal{E}$-matching problems.
- **Extended higher-order abstract syntax** enriches the higher-order abstract-syntax of Pfenning and Elliott [66]. With this abstract syntax, we show that extended matching can be seen as a generic framework for different application areas, particularly for program transformations.

In Chapter 2 we discuss the matching problems for a class of *first-order constructor-based specifications*, which is also a prerequisite for extended higher-order matching discussed in chapter 4. A concrete matching algorithm based on *structural narrowing* is developed, which is proved to be complete and produces no redundant matches. The termination of the algorithm for a class of *acyclic matching combinators* is proved. To improve the efficiency, three techniques are discussed: dependency analysis, partial sharing analysis, and commutative matching. Examples are presented to show the power of our approach.

In chapter 3 *contextual matching* is introduced. The problem comes from conditional term rewriting systems or specifications with conditional equations. Generally, contextual matching solves matching problems with extra conditions — contexts, which are collections of equations and disequations. To solve a matching problem with a context, we have to prove additionally whether the context can be satisfied under the matches found. The matching algorithm presented in chapter 2 is thus extended. Our goal is to reduce the number of matches and to speed up the matching process through using context information. Our interest here is more in the solution of technical problems than in theoretical results. Examples show that contextual matching is very useful.

Chapter 4 can be seen as a higher-order lifting of chapter 2. At first, a class of *higher-order constructor-based specifications* is given, and a matching algorithm is developed. Fortunately, all the nice properties of the first-order matching algorithm, such as completeness, soundness and termination, do also hold for the higher-order case. The applications are shown through examples. The theoretical results about higher-order rewriting and higher-order unification made in recent years provide important prerequisites for this work. Among them are higher-order pattern unification [55, 60], higher-order rewriting [58, 59, 69], and higher-order narrowing [70, 1].

In chapter 5 some implementation issues are discussed. We first enrich the higher-order abstract syntax in [66] to *extended higher-order abstract syntax*. Concretely, we give an example program language as an object language, and show how to represent transformations of object programs using the abstract syntax. At last, we give a brief review of the implementation structures of the matching algorithms.

Finally, chapter 6 contains conclusions and remarks future researches.

Chapter 2

Extended First-Order Matching

$\mathcal{E}$-unification, as a generalization of unification for uninterpreted terms, was first discussed by Plotkin in 1972, see [68]. Since then some well known positive results for associativity, commutativity [81], for boolean rings [53], and so on, demonstrate the importance of $\mathcal{E}$-unification in many application fields, like theorem proving, logic programming, etc. Enumerating a complete set of unifiers in an arbitrary equational theory is called "universal unification" by Siekmann [77]. Fay [21] was the first to give a non-trivial (not necessarily terminating) procedure for universal unification, called "narrowing", and to prove its completeness for any theory presented by a finite convergent rewriting system. The unification procedure obtained by narrowing usually does not terminate, and the complete set of unifiers generated is still very redundant. For these reasons, most of the research in narrowing has been concerned with finding methods to prune the search space.

In this chapter, we first introduce a class of constructor-based specifications which can be considered as convergent term rewriting systems. Then a concrete matching algorithm is developed, which is complete and produces no redundant matches for this class of specifications. To improve the efficiency of the algorithm, three techniques are discussed: dependency analysis, partial sharing analysis, and commutative matching. The termination of the algorithm for a restricted class of acyclic constructor-based functions is then proved. The comparison to second-order matching [40] and rewriting modulo commutativity sheds some light on the power of our matching algorithm. At last, we discuss some related work.

This chapter is also a prerequisite for extended higher-order matching discussed in chapter 4.

2.1 Constructor-Based Specifications

Constructor-based specifications are introduced to define a class of functions, called matching combinators. They are closely related to recursive functional programs, such as in the functional language ML, and are particular useful in defining syntactical similarities of programs.

The concatenation function $++$ on lists is an example of a matching combinator, which is defined by the following two equations:

$$\begin{array}{lcl} nil \mathbin{++} y & = & y \\ (x :: xs) \mathbin{++} y & = & x :: (xs \mathbin{++} y), \end{array}$$

where nil (empty list) and :: (prepending an element to a list) are two constructors of lists.

Using the function ++, we can define several useful program schemata, such as a sequence of expressions that contains a special element: $x \mathbin{++}(c :: nil) \mathbin{++} y$. Let the following be a program transformation rule:

$$x \mathbin{++}(c :: nil) \mathbin{++} y \Longrightarrow x \mathbin{++} y.$$

If we want to delete an expression c from a sequence of expressions e_s, an important thing should be done is to find sequences of expressions that can be substituted for the variables x and y, such that the equation $x \mathbin{++}(c :: nil) \mathbin{++} y = e_s$ holds. An obstacle to find such a substitution is the fact that ++ is not a free symbol, but defined by a set of first-order equations. Solving such kind of equations is a typical problem of extended first-order matching, which will be discussed in this chapter.

2.1.1 Preliminaries

In this subsection, some notation and terminology about first-order specifications and semantic matching will be introduced. The following is a basic example from classic many-sorted algebraic specifications, which defines the sort nat and the functions $+$, $-$, $*$, and max.

$$\begin{array}{lll} Nat & = & \\ & \textbf{sorts}: & nat \\ & \textbf{cons}: & 0 \ : \ nat \\ & & succ \ : \ nat \rightarrow nat \\ & \textbf{funs}: & +,\ -,\ *,\ max \ : \ nat \rightarrow nat \rightarrow nat \\ & \textbf{eqns}: & \forall\, x, y \in nat: \\ & & 0 + y = y \\ & & succ(x) + y = succ(x + y) \\ & & \\ & & 0 - y = 0 \\ & & succ(x) - 0 = succ(x) \\ & & succ(x) - succ(y) = x - y \\ & & \\ & & 0 * y = 0 \\ & & succ(x) * y = y + (x * y) \\ & & \\ & & max(0, y) = y \\ & & max(succ(x), 0) = succ(x) \\ & & max(succ(x), succ(y)) = succ(max(x, y)) \end{array}$$

Signatures

Definition 2.1 (Signatures) A signature $\Sigma = \langle \mathcal{S}, \mathcal{C}, \mathcal{F} \rangle$ consists of

- a set $\mathcal{S}$ of **sorts**;
- a set $\mathcal{C}$ of **constructor symbols** generating terms; and
- a set $\mathcal{F}$ of **function symbols** manipulating terms.

In the example, $\mathcal{S} = \{nat\}$, $\mathcal{C} = \{0, succ\}$, and $\mathcal{F} = \{+, -, *, max\}$. Each constructor symbol or function symbol has a **type** $s_1 \to \cdots \to s_n \to s$ with $s_1, \cdots, s_n, s \in \mathcal{S}$, where $n \geq 0$ is its **arity**. For instance, the type of 0 is nat with arity 0; the type of $succ$ is $nat \to nat$ with arity 1; and $+$, $-$, $*$ and max have the type $nat \to nat \to nat$ with arity 2. Functions or constructors of 0-arity are often called **constants**.

Let Σ be a signature and $\mathcal{V}_s$ a set of **variables** of sort s for each sort s in Σ. We assume that these sets $\mathcal{V}_s$ are pairwise disjoint and also disjoint to $\mathcal{C}$ and $\mathcal{F}$. Then the union

$$\mathcal{V} = \bigcup_{s \in \mathcal{S}} \mathcal{V}_s$$

is called a set of **variables with respect to** Σ.

Definition 2.2 (Terms) Let Σ be a signature and $\mathcal{V}$ be a set of variables with respect to Σ. The set $\mathcal{T}_s(\mathcal{V})$ of terms of sort s is inductively defined by:

- $\mathcal{V}_s \subseteq \mathcal{T}_s(\mathcal{V})$;
- if $f \in \mathcal{C} \cup \mathcal{F}$ and has the type $s_1 \to \cdots \to s_n \to s$ with $n \geq 0$, $t_1 \in \mathcal{T}_{s_1}(\mathcal{V})$, $\cdots$, $t_n \in \mathcal{T}_{s_n}(\mathcal{V})$, then $(f\ t_1 \cdots t_n) \in \mathcal{T}_s(\mathcal{V})$;
- no other terms of sort s are in $\mathcal{T}_s(\mathcal{V})$.

The set $\mathcal{T}_s = \mathcal{T}_s(\emptyset)$ is the set of terms without variables, called **ground terms**. The sets

$$\mathcal{T}(\mathcal{V}) = \bigcup_{s \in \mathcal{S}} \mathcal{T}_s(\mathcal{V}) \quad \text{and} \quad \mathcal{T} = \bigcup_{s \in \mathcal{S}} \mathcal{T}_s$$

are the **set of all terms** and the **set of all ground terms** respectively.

For instance, 0, $(succ\ 0)$, $(succ(succ\ 0))$, and $((succ\ 0) + (succ\ 0))$ are all ground terms of sort nat. If x and y are variables of sort nat, then x, $(succ\ x)$, $(succ(succ\ x))$, $(x + y)$, and $(x * y)$ are all in $\mathcal{T}_{nat}(\{x, y\})$. Note that some constructor and unction symbols are used as infix symbols, such as ::, $+$, $-$ and $*$.

The reason for using the curried notations $s_1 \to \cdots \to s_n \to s$ and $(f\ t_1 \cdots t_n)$, instead of $s_1 \times \cdots \times s_n \to s$ and $f(t_1, \cdots, t_n)$ as usual, is the easy and compatible extension of higher-order signatures.

Convention:

- The set of variables in a term t is denoted as $\mathcal{V}(t)$.
- Some parentheses will be omitted if there is no confusion, for example, we write $succ\ 0$ instead of $(succ\ 0)$.

A term t in $\mathcal{T}(\mathcal{V})$ can also be viewed as a finite ordered tree. Its leaves are labeled with variables or constants; its internal nodes are labeled with constructor or function symbols and have an outdegree equal to the arity of the label. A **position** or an **occurrence** within a term may be represented – in Dewey decimal notation – as a sequence of positive integers, describing the path from the outermost "root" symbol to the head of the subterm at that position. The set $\mathcal{O}(t)$ of occurrences in a term t is inductively defined as

$$\begin{array}{lcll} \mathcal{O}(a) & = & \{\Lambda\}, & \text{if } a \text{ is a variable or a constant} \\ \mathcal{O}(f\ t_1 \cdots t_n) & = & \{\Lambda\} \cup \{i.p | 1 \leq i \leq n \wedge p \in \mathcal{O}(t_i)\}, & \text{if } f \text{ is a constructor or a function.} \end{array}$$

By $t|_p$ we denote the **subterm** of t rooted at position p. For example, if

$$t = (succ\ x) + (succ(succ\ y)),$$

then $t|_{2.1}$ is the first subterm of t's second subterm, which is $succ\ y$. The position of "root" is denoted as Λ. We speak of a position p as being **above** a position q in some term t if p (represented as a sequence of numbers) is a prefix of q, i.e., if occurrence $t|_q$ is within $t|_p$. Two positions p and q are called **disjoint** if p is not above q and vice versa. A subterm of t is called **proper** if it is distinct from t. The **replacement** of a subterm of a term t at position p with a term s is denoted by $t[s]_p$. This can be extended to several positions: $t[s_1, \cdots, s_k]_{\{p_1, \cdots, p_k\}}$ means replacing subterms at pairwise disjoint positions p_1, $\cdots$, p_k with terms s_1, $\cdots$, s_k simultaneously. The set of non-variable occurrences in a term t is defined as:

$$\overline{\mathcal{O}}(t) = \{p \in \mathcal{O}(t) \mid t|_p \notin \mathcal{V}\}.$$

A **substitution** σ is a map from $\mathcal{V}$ to $\mathcal{T}(\mathcal{V})$ that differs from the identity only in finitely many places and is presented as a finite set of ordered pairs of the form

$$\sigma = \{x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\}.$$

The **domain** and **range** of σ are $\mathcal{D}(\sigma) = \{x_1, \cdots, x_n\}$ and $\mathcal{I}(\sigma) = \mathcal{V}(t_1) \cup \cdots \mathcal{V}(t_n)$ respectively. For any term t, σt is inductively defined by:

$$\begin{array}{lcll} \sigma x_i & = & t_i, & \text{if } 1 \leq i \leq n \\ \sigma x & = & x, & \text{if } x \notin \{x_1, \cdots, x_n\} \\ \sigma(f\ t_1 \cdots t_n) & = & f\ (\sigma t_1) \cdots (\sigma t_n), & \text{if } n \geq 0. \end{array}$$

A substitution σ is a **renaming** if it is a function from $\mathcal{V} \rightarrow \mathcal{V}$.

Definition 2.3 (Idempotence) A substitution σ is idempotent if $\sigma \circ \sigma = \sigma$.

A sufficient condition for idempotence is given by Snyder in [78]: A substitution σ is idempotent whenever $\mathcal{I}(\sigma) \cap \mathcal{D}(\sigma) = \emptyset$.

Convention: Without loss of generality, we will in most contexts restrict our attention to idempotent substitutions. For a justification of this convention see [78].

The **composition** of substitutions σ and θ is the substitution, denoted by $\sigma \circ \theta$, such that for every variable x $(\sigma \circ \theta)x = \sigma(\theta x)$. Given a set W of variables, we say that two substitutions σ and θ are **equal over** W, denoted as $\sigma =_{[W]} \theta$, iff $\sigma x = \theta x$ for all $x \in W$. We say σ is **more general than** θ **over** W, denoted as $\sigma \leq_{[W]} \theta$, iff there exists some ρ such that $\theta =_{[W]} \rho \circ \sigma$.

Specifications

Definition 2.4 (Specifications) A specification $\mathcal{SP} = \langle \Sigma, \mathcal{E} \rangle$ consists of

- a signature Σ;
- a set $\mathcal{E}$ of **equations** of the form $l = r$, where l and r are two terms of the same sort.

Let $\mathcal{SP} = \langle \langle \mathcal{S}, \mathcal{C}, \mathcal{F} \rangle, \mathcal{E} \rangle$ be a specification, and $\mathcal{SP}_1 = \langle \langle \mathcal{S}_1, \mathcal{C}_1, \mathcal{F}_1 \rangle, \mathcal{E}_1 \rangle$ such that $\mathcal{S}_1 \cap \mathcal{S} = \emptyset$, $\mathcal{C}_1 \cap \mathcal{C} = \emptyset$, and $\mathcal{F}_1 \cap \mathcal{F} = \emptyset$. The **extension of $\mathcal{SP}$ with $\mathcal{SP}_1$**, denoted as $\mathcal{SP} + \mathcal{SP}_1$ is the following specification:

$$\langle \langle \mathcal{S} \cup \mathcal{S}_1, \mathcal{C} \cup \mathcal{C}_1, \mathcal{F} \cup \mathcal{F}_1 \rangle, \mathcal{E} \cup \mathcal{E}_1 \rangle .$$

Usually, specifications can be parameterized by other specifications. To avoid technical details, which are of little interest for solving matching problems, we will only consider specifications parameterized by sorts. A **parameterized specification** is usually denoted as $\mathcal{SP}[s_1, \cdots, s_n] = \langle \Sigma, \mathcal{E} \rangle$, where s_1, $\cdots$, s_n are called **formal parameters**. For instance, *Pair* is a frequently used parameterized specification.

$$\begin{array}{rll} \mathit{Pair}[\alpha_1, \alpha_2] & = & \\ & \textbf{sorts}: & \mathit{pair} \\ & \textbf{cons}: & (,) \ : \ \alpha_1 \to \alpha_2 \to \mathit{pair} \\ & \textbf{funs}: & \mathit{fst} \ : \ \mathit{pair} \to \alpha_1 \\ & & \mathit{snd} \ : \ \mathit{pair} \to \alpha_2 \\ & \textbf{eqns}: & \forall\, x \in \alpha_1; \ y \in \alpha_2 : \\ & & \mathit{fst}(x, y) = x \\ & & \mathit{snd}(x, y) = y \end{array}$$

List$[s]$ is another well-known parameterized specification, which is also an extension of *Nat*.

$$\begin{array}{rll} \mathit{List}[\alpha] & = \ \mathit{Nat} + & \\ & \textbf{sorts}: & \mathit{list} \\ & \textbf{cons}: & \mathit{nil} \ : \ \mathit{list} \\ & & :: \ : \ \alpha \to \mathit{list} \to \mathit{list} \\ & \textbf{funs}: & +\!\!+ \ : \ \mathit{list} \to \mathit{list} \to \mathit{list} \\ & & \mathit{len} \ : \ \mathit{list} \to \mathit{nat} \\ & & \mathit{hd} \ : \ \mathit{list} \to \alpha \\ & & \mathit{tl} \ : \ \mathit{list} \to \mathit{list} \\ & \textbf{eqns}: & \forall\, x \in \alpha; \ xs, ys \in \mathit{list} : \\ & & \mathit{nil} +\!\!+ ys = ys \\ & & (x :: xs) +\!\!+ ys = x :: (xs +\!\!+ ys) \\ & & \\ & & \mathit{len}\ \mathit{nil} = 0 \\ & & \mathit{len}(x :: xs) = \mathit{succ}(\mathit{len}\ xs) \\ & & \\ & & \mathit{hd}(x :: xs) = x \\ & & \mathit{tl}(x :: xs) = xs \end{array}$$

If we want to instantiate a parameterized specification in a context more than once, we have to **rename** the sorts introduced in its body. We use the notation

$$List[nat]\{list \textbf{ with } list\_nat\}$$

to denote the specification by substituting the formal sort s in *List* with *nat*, and the sort *list* is renamed to *list_nat*.

The set of equations $\mathcal{E}$ in a specification induces a **congruence relation** $=_\mathcal{E}$ over terms:

$$\begin{array}{lll}
l =_\mathcal{E} r, & \text{if } l = r \in \mathcal{E} & \text{(axiom)} \\
\sigma t =_\mathcal{E} \sigma s, & \text{if } t =_\mathcal{E} s \text{ and } \sigma \text{ is a substitution} & \text{(instance)} \\
t =_\mathcal{E} t, & & \text{(reflexivity)} \\
s =_\mathcal{E} t, & \text{if } t =_\mathcal{E} s & \text{(symmetry)} \\
t_1 =_\mathcal{E} t_3, & \text{if } t_1 =_\mathcal{E} t_2 \text{ and } t_2 =_\mathcal{E} t_3 & \text{(transitivity)} \\
f\ t_1 \cdots t_n =_\mathcal{E} f\ s_1 \cdots s_n, & \text{if } t_1 =_\mathcal{E} s_1, \cdots, t_n =_\mathcal{E} s_n & \text{(congruence)}
\end{array}$$

Term rewriting systems

Term rewriting systems are a special class of specifications: instead of equations, one uses rules that are oriented pairs of terms.

Definition 2.5 (Term rewriting systems) A term rewriting system $\mathcal{TRS} = \langle \Sigma, \mathcal{R} \rangle$ consists of

- a signature Σ;
- a set $\mathcal{R}$ of **rewrite rules** of the form $l \rightarrow r$, where l and r are of the same sort. In addition, l is not a variable and $\mathcal{V}(r) \subseteq \mathcal{V}(l)$.

Treating the equations in the specification *Nat* as rewrite rules from left to right, we get a rewriting system, say *TNat*.

Let $\mathcal{TRS} = \langle \Sigma, \mathcal{R} \rangle$ be a term rewriting system. A rule $l \rightarrow r$ is called **left linear** if no variables occur more than once in l; $\mathcal{TRS}$ is called **left linear** if every rule in $\mathcal{R}$ is left linear. *TNat* is a left linear system. A rule $l \rightarrow r$ is called **variable-preserving** if $\mathcal{V}(r) = \mathcal{V}(l)$, otherwise it is **variable-dropping**. For instance, $(succ\ x) + y \rightarrow succ(x + y)$ is variable-preserving, but $0 - y \rightarrow 0$ is variable-dropping. $\mathcal{TRS}$ is called variable-preserving if every rule in $\mathcal{R}$ is variable-preserving, otherwise it is variable-dropping. So *TNat* is a variable-dropping system. Two rules $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ are called **overlapping**, if there exist an occurrence $p \in \overline{\mathcal{O}}(l_2)$, a renaming θ, and a substitution σ such that $\sigma(\theta l_1)$ and $\sigma(l_2|_p)$ are equal.

Different from equations, the set of rewrite rules $\mathcal{R}$ defines a **rewriting relation** $\rightarrow_\mathcal{R}$. $t \rightarrow_\mathcal{R} s$ reads "t $\mathcal{R}$-**rewrites to** s in one step", i.e. there exist a position p in t, a substitution σ, and a rule $l \rightarrow r \in \mathcal{R}$ such that

- $\sigma l = t|_p$, and
- $s = t[\sigma r]_p$.

A **rewriting sequence** is a concatenation of rewriting steps $t_0 \rightarrow_{\mathcal{R}} t_1 \rightarrow_{\mathcal{R}} t_2 \rightarrow_{\mathcal{R}} \cdots$. As usual, $\rightarrow^*_{\mathcal{R}}$ denotes the reflexive and transitive closure of $\rightarrow_{\mathcal{R}}$. For instance,

$$(succ\ 0) + (succ(succ\ 0)) \rightarrow_{TNat} succ(0 + (succ(succ\ 0))) \rightarrow_{TNat} succ(succ(succ\ 0)),$$

$$\text{or } (succ\ 0) + (succ(succ\ 0)) \rightarrow^*_{TNat} succ(succ(succ\ 0)).$$

A term t is $\mathcal{R}$-**irreducible** or an $\mathcal{R}$-**normal form** iff there exists no term s such that $t \rightarrow_{\mathcal{R}} s$. Two terms t_1 and t_2 are called **joinable**, denoted as $t_1 \downarrow_{\mathcal{R}} t_2$, iff there exists a term t' such that $t_1 \rightarrow^*_{\mathcal{R}} t'$ and $t_2 \rightarrow^*_{\mathcal{R}} t'$. For instance, $(succ\ 0) + (succ(succ\ 0)) \downarrow (succ(succ\ 0)) + (succ\ 0)$.

Definition 2.6 Let $\mathcal{TRS} = \langle \Sigma, \mathcal{R} \rangle$ be a term rewriting system.

- $\mathcal{TRS}$ is **strongly normalizing** if there is no infinite rewriting sequence $t_0 \rightarrow_{\mathcal{R}} t_1 \rightarrow_{\mathcal{R}} t_2 \rightarrow_{\mathcal{R}} \cdots$.
- $\mathcal{TRS}$ is **confluent**, or **Church-Rosser**, if $t \rightarrow^*_{\mathcal{R}} t_1$ and $t \rightarrow^*_{\mathcal{R}} t_2$, then $t_1 \downarrow_{\mathcal{R}} t_2$.
- $\mathcal{TRS}$ is **convergent** if it is confluent and strongly normalizing.
- $\mathcal{TRS}$ is **orthogonal** if it is left linear and the rules in $\mathcal{R}$ are pairwise non-overlapping.

TNat is strongly normalizing and confluent, so it is convergent; it is also orthogonal.

Convention: We will omit the index $\mathcal{R}$, if there is no confusion.

Discussing extended matching problems, one should consider specifications in general. However, term rewriting systems, as a subclass of specifications, will be discussed in this thesis, for the following two reasons:

- To obtain the desirable properties of extended matching, such as completeness and termination.
- To be practically useful, i.e., it should be easily and efficiently implementable.

Semantic matching and unification

Let us discuss semantic matching and semantic unification through examples. Consider the terms $t = x + y$ and $s = succ(succ(succ\ 0))$ of sort *nat*. Typical problems to be decided by **semantic matching** are:

- Is there a substitution σ such that $\sigma t =_{\mathcal{E}} s$?
- How many such substitutions are there? – exactly one, finitely many, or infinitely many?

For our example, there are four substitutions:

$$\begin{aligned}
\sigma_1 &= \{x \mapsto succ(succ(succ\ 0)), y \mapsto 0\} \\
\sigma_2 &= \{x \mapsto succ(succ\ 0), y \mapsto succ\ 0\} \\
\sigma_3 &= \{x \mapsto succ\ 0, y \mapsto succ(succ\ 0)\} \\
\sigma_4 &= \{x \mapsto 0, y \mapsto succ(succ(succ\ 0))\}
\end{aligned}$$

Semantic unification, as a generalization of semantic matching, solves the following problems (let t and s be two terms):

- Are there terms that can be substituted for the variables in t and s, such that $\sigma t =_{\mathcal{E}} \sigma s$?
- How many such substitutions are there?

In [34], it is proved that there exist some convergent term rewriting systems that have decidable semantic matching but undecidable semantic unification. *TNat* is such an example, see also [8, 17].

Let $\mathcal{SP} = \langle \Sigma, \mathcal{E} \rangle$ be a specification. A **matching pair** or a **unification pair** of SP is an ordered pair of terms (t, s) with t and s terms of Σ. A **match** or a **unifier with respect to** $\mathcal{SP}$ of the pair is a substitution σ such that $\sigma t =_{\mathcal{E}} s$ and $\sigma t =_{\mathcal{E}} \sigma s$ respectively. A **matching** or a **unification problem** of $\mathcal{SP}$ is a set of matching or unification pairs $\{(t_1, s_1), \cdots, (t_n, s_n)\}$. A **match** or a **unifier with respect to** $\mathcal{SP}$ of the problem is a substitution σ such that $\sigma t_i =_{\mathcal{E}} s_i$ and $\sigma t_i =_{\mathcal{E}} \sigma s_i$ $(1 \leq i \leq n)$ respectively. A matching problem $\{(t_1, s_1), \cdots, (t_n, s_n)\}$ is called **ground** if all s_i $(1 \leq i \leq n)$ are ground terms.

The **set of all matches** of Γ will be denoted as $\mathcal{M}_{\mathcal{SP}}(\Gamma)$. For most applications, one does not need the set of all matches.

Definition 2.7 (Complete set of matches) Let $\Gamma = \{(t_1, s_1), \cdots, (t_n, s_n)\}$ be a matching problem of a specification $\mathcal{SP} = \langle \Sigma, \mathcal{E} \rangle$. $\mathcal{CSM}_{\mathcal{SP}}(\Gamma)$ is a complete set of matches of Γ with respect to $\mathcal{SP}$ iff

$$\begin{array}{lll} (1) & \mathcal{CSM}_{\mathcal{SP}}(\Gamma) \subseteq \mathcal{M}_{\mathcal{SP}}(\Gamma) & \text{(consistence)} \\ (2) & \forall \sigma \in \mathcal{M}_{\mathcal{SP}}(\Gamma).\ \exists \theta \in \mathcal{CSM}_{\mathcal{SP}}(\Gamma).\ \theta \leq_{\mathcal{E}} \sigma & \text{(completeness)} \end{array}$$

$\mathcal{CSMM}_{\mathcal{SP}}(\Gamma)$ is a **complete set of minimal matches** of Γ with respect to $\mathcal{SP}$ iff it satisfies an additional condition

$$(3) \quad \forall \sigma, \sigma' \in \mathcal{CSMM}_{\mathcal{SP}}(\Gamma).\ \sigma \not\leq_{\mathcal{E}} \sigma', \text{if } \sigma \neq \sigma' \quad \text{(minimality)}$$

Similarly, we can define the set of all unifiers $\mathcal{U}_{\mathcal{SP}}(\Gamma)$; a complete set of unifiers $\mathcal{CSU}_{\mathcal{SP}}(\Gamma)$, and the complete set of minimal unifiers $\mathcal{CSMU}_{\mathcal{SP}}(\Gamma)$.

For example, the set $\{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$ in the above example forms the complete set of minimal matches of $\{(x + y,\ succ(succ(succ\ 0)))\}$.

Convention: If there is no confusion, we will omit the index $\mathcal{SP}$.

Narrowing

In general, narrowing is a technique used to solve equations. In particular, it is often used in the context of semantic unification and semantic matching. Recently there has been much interest in incorporating the logic and the functional programming paradigms in a single language. Narrowing is the computational mechanism underlying many of these amalgamated languages. Examples include ALF [30], EQLOG [27] and SLOG [22].

Definition 2.8 (Narrowing) We say that a term t is **narrowable** into a term s if there exist a position p in t, a renaming $l' \to r'$ of a rule $l \to r \in \mathcal{R}$, and a substitution σ such that

- σ is the most general unifier of $t|_p$ and l',

- $s = \sigma(t[r']_p)$,

denoted as $t \rightsquigarrow_{p,l\to r,\sigma} s$, or $t \rightsquigarrow_{l\to r} s$ shortly.

We write $t \rightsquigarrow^* t'$ if there exists a **narrowing derivation**

$$t = t_1 \rightsquigarrow_{p_1,l_1\to r_1,\sigma_1} t_2 \rightsquigarrow_{p_2,l_2\to r_2,\sigma_2} \cdots \rightsquigarrow_{p_{n-1},l_{n-1}\to r_{n-1},\sigma_{n-1}} t_n = t'.$$

Fay and Hullot showed that narrowing is a complete method for solving equations in a convergent term rewriting system, see [21] and [41]. Completeness means that for every solution to a given problem, a more general solution can be found by narrowing. In order to reduce the search space of narrowing, Hullot introduced the concept of basic narrowing in [41] and showed that basic narrowing is complete for convergent rewriting systems. There are some other narrowing strategies for choosing a subterm to apply narrowing. Among them are innermost narrowing [22] and lazy narrowing [86].

2.1.2 Constructor-Based Specifications

For program transformations, a class of equational specifications – constructor-based specifications – is particularly interesting, which can be treated as term rewriting systems. So from now on, we do not distinguish term rewriting systems and specifications.

Convention:

- If we say 'a specification', we mean the term rewriting system obtained by treating the equations as rewrite rules from left to right.
- A term t is an $\mathcal{E}$-normal form if it is a normal form with respect to the rewriting system.

Semantically, $t_1 =_{\mathcal{E}} t_2$ if and only if there exists an $\mathcal{E}$-normal form t such that $t_1 =_{\mathcal{E}} t$ and $t_2 =_{\mathcal{E}} t$.

Definition 2.9 (Constructor-based specifications) A specification $\mathcal{SP} = \langle \Sigma, \mathcal{E} \rangle$ is called constructor-based, if the following conditions are satisfied:

- $\mathcal{SP}$ is an orthogonal specification;
- if $l = r \in \mathcal{E}$, then l has the form $f\ l_1 \cdots l_n$ with f a function symbol in Σ, and each l_i $(1 \leq i \leq n)$ contains only variables and constructors.

Functions in constructor-based specifications are called **matching combinators** (shortly: **combinators**). *Nat* and *List* are both examples of constructor-based specifications. $\mathcal{E}_f$ denotes the set of equations associated with function f, so

$$\mathcal{E}_+ \ = \ \{0 + y = y,\ (succ\ x) + y = succ(x + y)\}.$$

From the definition, we can see that equations in constructor-based specifications are used to define functions. They can naturally be treated as rewrite rules from left to right. This kind of function definitions is widely used in functional programming languages.

Proper non-decreasing functions

A **partial ordering** is an irreflexive, transitive binary relation. For example, the greater-than relation on natural numbers is a partial ordering. As usual, if $\succ$ is a partial ordering, its reflexive closure is denoted as $\succeq$: $t \succeq s$ means $t \succ s$ or $t = s$. The greater-than-or-equal-to relation on the natural numbers is a reflexive closure of the greater-than relation. A partial ordering is **well-founded** if there are no infinite descending sequences $t_0 \succ t_1 \succ t_2 \succ \cdots$. Thus, the greater-than relation is a well-founded partial ordering on natural numbers.

Let $\succ$ be a partial ordering on S, then **multiset ordering** $\succ_{mul}$ is a partial ordering on the multisets over S. $X = \{t_1, \cdots, t_n\} \succ_{mul} Y = \{s_1, \cdots, s_m\}$, if $X \neq Y$, and for any $s_i \in Y - X$ $(1 \leq i \leq m)$, there exists a $t_j \in X - Y$ $(1 \leq j \leq n)$ such that $t_j \succ s_i$. It is shown in [14] that the multiset ordering is well-founded iff the ordering $\succ$ is well-founded. The definition of well-founded partial ordering suggests that it is useful in termination proofs.

Definition 2.10 (Suitable property) A suitable property P is a measure associated with ground terms, along with a well-founded partial ordering $\rhd$ that compares results of P, such that $P(t) \rhd P(s)$ if s is a proper subterm of t.

Let t be any term, then the **depth** of t, which is defined by the following equations, is a suitable property along with the greater-than relation $>$ on natural numbers.

$$\begin{array}{lcll} depth(a) & = & 0, & \text{if } a \text{ is a variable or a constant} \\ depth(f\ t_1 \cdots\ t_n) & = & \max_{1 \leq i \leq n} depth(t_i) + 1, & n \geq 0. \end{array}$$

Let P be a suitable property, there may exist some functions whose results are "less than" some arguments with respect to P. We take $-$ as an example:

$$(succ\ 0) - (succ\ 0) = 0$$

but $depth(succ\ 0) > 0$. That is, the result 0 is "less than" both arguments $succ\ 0$ with respect to $depth$.

Definition 2.11 (Non-decreasing function) Let P be a suitable property along with the well-founded partial ordering $\rhd$, $\langle \Sigma, \mathcal{E} \rangle$ a constructor-based specification. A function symbol f in Σ is called non-decreasing (with respect to P) if whenever $f\ s_1 \cdots s_n =_{\mathcal{E}} t$, where s_i $(1 \leq i \leq n)$ and t are all ground $\mathcal{E}$-normal forms, then $P(s_i) \not\rhd P(t)$. Any function which does not have this property is said to be **decreasing** (with respect to P).

It is decidable whether a function defined by a constructor-based specification is non-decreasing with respect to a property P or not. It is not true for functions defined by general term rewriting systems, see [17].

The above example shows that $-$ is a decreasing function with respect to $depth$. However, it can be shown that $+$, max and $++$ are all non-decreasing functions with respect to $depth$. We take $+$ as an example. Let $a_1 + a_2 =_{\mathcal{E}_+} a$, where a_1, a_2 and a are all ground $\mathcal{E}$-normal forms; we show that $depth(a_1) \not> depth(a)$ and $depth(a_2) \not> depth(a)$:

- if a_1 is 0, then $a_1 + a_2 =_{\mathcal{E}_+} a_2$, we have $depth(a_1) \not> depth(a_2)$ and $depth(a_2) \not> depth(a_2)$;

- if $a_1 = succ\ a_1'$, then $a_1 + a_2 =_{\mathcal{E}_+} succ(a_1' + a_2)$. Let $a_1' + a_2 =_{\mathcal{E}_+} a'$, by the induction hypothesis $depth(a_1') \not> depth(a')$ and $depth(a_2) \not> depth(a')$, so $depth(a_1) \not> depth(succ\ a')$ and $depth(a_2) \not> depth(succ\ a')$. □

The definition of the non-decreasing property is only appropriate for functions that are defined by variable-preserving equations, such as $+$, *max* and $++$. For functions containing variable-dropping equations, such as $*$ in *Nat* and *len* in *List*, the definition is not appropriate, since some parameters of such functions do not affect the results. After the definition, $*$ and *len* are decreasing functions with respect to *depth*. This can be shown by the following example,

$$len((succ(succ\ 0)) :: nil) =_{\mathcal{E}} succ\ 0,$$

but $depth(succ(succ\ 0)) > depth(succ\ 0)$. For variable-dropping functions, we should generalize the definition.

Definition 2.12 (Proper non-decreasing functions) Let P be a suitable property along with the well-founded partial ordering $\rhd$, $\langle \Sigma, \mathcal{E} \rangle$ a constructor-based specification. A function symbol f in Σ is defined to be proper non-decreasing (with respect to P) if whenever σ is a minimal match of $f\ t_1 \cdots\ t_n$ and t, where each t_i is in normal form and contains no function symbols $(1 \leq i \leq n)$, t is a normal form, then $P(\sigma y) \not\rhd P(t)$ for all $y \in \mathcal{D}(\sigma)$, by assuming $P(x) \not\rhd P(a)$ for any variable x and ground term a. Any function that does not have this property is said to be **proper decreasing** (with respect to P).

The concepts "suitable property" and "non-decreasing" are used in some papers, such as in [17]. "Proper non-decreasing" is a new concept introduced here to treat functions containing variable-dropping equations, which is important to define a class of acyclic matching combinators (see below). Now we can show inductively that $*$ and *len* are proper non-decreasing with respect to *depth*. Take *len* as an example. Let σ be a minimal match of $len\ t$ and a.

- If a is 0, then t is *nil*, or a variable x and $\sigma x = nil$. We have $depth(\sigma x) \not> depth(0)$.
- If $a = succ\ n$, then t
  - has the form $t_1 :: t_s$ and $len\ t_s = n$. This case can be proved according to the induction hypothesis;
  - or is a variable x and $\sigma x = x_0 :: (x_1 :: (\cdots (x_n :: nil) \cdots))$, where all x_i $(0 \leq i \leq n)$ are variables. So, $depth(\sigma x) \not> depth(succ\ n)$. □

Acyclic matching combinators

With all these preparations, we can now define a class of functions which have finite semantic matching problems. That is, if Γ is a matching problem in which all functions are in this class, then Γ has a finite complete set of matches.

Definition 2.13 (Acyclic matching combinators) Let P be some suitable property, $\langle \Sigma, \mathcal{E} \rangle$ a constructor-based specification, and $f \in \Sigma$. f is called an acyclic matching combinator (short: **acyclic combinator**), if for any $l = r \in \mathcal{E}_f$, the following conditions are satisfied:

- r has the form $a\ r_1 \cdots r_n$ with a a variable or a constant, $n \geq 0$;
- if g is another function symbol in r, then g is an acyclic combinator; and
- there is no function symbol in r nested below any function that is proper decreasing with respect to P.

Obviously, $+$, max, $++$, and len are all acyclic matching combinators, but $*$ and $-$ are not, because they contain equations $succ(x) * y = y + (x * y)$ and $succ(x) - succ(y) = x - y$ respectively. The following set of equations also defines the function $*$, but now $*$ becomes an acyclic combinator. In the sequel, if we use $*$, we mean this new definition.

$$\begin{array}{lcl} 0 * y & = & 0 \\ (succ\ x) * 0 & = & 0 \\ (succ\ x) * (succ\ y) & = & succ(y + (x * (succ\ y))) \end{array}$$

There is no possible way to change $-$ to an acyclic combinator, at least to our knowledge.

2.2 The Matching Algorithm $\mathcal{MA}^1$

To solve a matching problem with respect to a specification, one needs a matching algorithm. Such an algorithm should be able to decide whether a given problem is solvable, and if the answer is 'yes', it should compute a finite complete set of matches of the problem. For reasons of efficiency, the complete set of matches computed by a matching algorithm should be as small as possible. At best, the algorithm computes a complete set of minimal matches without too much overhead of comparing between the matches.

Definition 2.14 (Solved form) A matching pair (x, t) in a matching problem Γ is in solved form, if x is a variable which does not occur anywhere else in Γ; x in this pair is called a **solved variable**. A matching problem is in solved form if all its matching pairs are in solved form.

Note that a matching problem in solved form can always be presented as an idempotent substitution.

Convention:

- Γ_σ denotes the solved matching problem $\{(x_1, t_1), \cdots, (x_n, t_n)\}$ of the substitution $\sigma = \{x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\}$.
- If $\Gamma = \{(x_1, t_1), \cdots, (x_n, t_n)\}$ is a matching problem in solved form, then it forms a substitution $\sigma_\Gamma = \{x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\}$.

The importance of solved forms is shown by the following lemma. A proof can be found in [80].

Lemma 2.1 Let $\Gamma = \{(x_1, t_1), \cdots, (x_n, t_n)\}$ be in solved form. If $\sigma = \{x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\}$, then σ is an idempotent most general match of Γ. Furthermore, for any match θ of Γ, we have $\theta = \theta \circ \sigma$. $\square$

2.2.1 Matching by Transformations on System $\mathcal{TS}^1$

Abstractly, a unification or matching process can be defined by a set of non-deterministic rules that transform a unification or matching problem into an explicit representation of its solutions, if the problem is solvable. This elegant approach is due to Martelli and Montanari (see [52]) and is widely used, such as in [79, 60]. This removal of control and data structure definition allows us to examine the fundamental properties of the problem more clearly.

The following is a transformation system to solve unification problems of specifications with empty sets of equations. In [78], Snyder has proved that this set of transformation rules can always find the unique most general unifier (short: mgu) of a unification problem, if it exists.

$$\{(t, t)\} \cup \Gamma \Longrightarrow \Gamma.$$

$$\{(f\ t_1 \cdots\ t_n, f\ s_1\ \cdots s_n)\} \cup \Gamma \Longrightarrow \{(t_1, s_1), \cdots, (t_n, s_n)\} \cup \Gamma,$$

if f is a constructor or function symbol.

$$\{(x, t)\} \cup \Gamma \Longrightarrow \{(x, t)\} \cup \{x \mapsto t\}(\Gamma), \qquad \text{if } x \in \mathcal{V}(\Gamma).$$

$$\{(t, x)\} \cup \Gamma \Longrightarrow \{(x, t)\} \cup \Gamma.$$

Convention: If Γ_1 and Γ_2 are two unification or matching problems, then $\Gamma_1 \cup \Gamma_2$ always denotes their disjoint union.

Structural narrowing

Instead of using an existing narrowing strategy, such as basic narrowing or innermost narrowing, we introduce a new narrowing strategy – structural narrowing. It is particularly appropriate to solve matching problems of constructor-based specifications and makes implementation easy and efficient.

Definition 2.15 (Structural narrowing) A non-variable term s is **structurally narrowable** into a term t **with** an equation $l = r$, if there exist a renaming $l' = r'$ of $l = r$ and a substitution σ, such that $\sigma = mgu(s, l')$ and $t = \sigma r'$, denoted also as $s \rightsquigarrow_{l=r,\sigma} t$, or $s \rightsquigarrow_{l=r} t$ for short. A term is called **structurally normalized**, if it is not structurally narrowable with any equation.

See the following examples:

$$\begin{array}{lll} xs \mathbin{++} ys & \rightsquigarrow_{nil ++ ys = ys} & ys_1 \qquad (1) \\ xs \mathbin{++} ys & \rightsquigarrow_{(x::xs) ++ ys = x::(xs ++ ys)} & x_1 :: (xs_1 \mathbin{++} ys_1) \qquad (2) \end{array}$$

In (1), the equation $nil \mathbin{++} ys = ys$ is renamed to $nil \mathbin{++} ys_1 = ys_1$ and the substitution is $\{xs \mapsto nil,\ ys \mapsto ys_1\}$; in (2) the equation $(x :: xs) \mathbin{++} ys = x :: (xs \mathbin{++} ys)$ is renamed to $(x_1 :: xs_1) \mathbin{++} ys_1 = x_1 :: (xs_1 \mathbin{++} ys_1)$ and the substitution is $\{xs \mapsto x_1 :: xs_1,\ ys \mapsto ys_1\}$.

In structural narrowing, the occurrence in s to be narrowed is always Λ. For example, $succ(x + y)$ is not structurally narrowable with $(succ\ x) + y = succ(x + y)$, since $+$ and $succ$ are different symbols; but $x - y$ is structurally narrowable with $(succ\ x) - (succ\ y) = x - y$.

Lemma 2.2 If t is not structurally narrowable with $l = r$, and σ is a substitution, then σt is also not structurally narrowable with the equation. $\square$

Proof. Suppose that σt is structurally narrowable with $l = r$ and l has no common variables with t and σ, otherwise we can rename the variables in $l = r$. So, t and l are also unifiable. $\square$

To make structural narrowing work, we should abstract any term into a term containing only one occurrence of a function symbol.

Definition 2.16 (Combinator abstractions) A combinator abstraction of a term t is denoted as $t|_{\{p_1,\cdots,p_k\}}$ such that

- $p_1, \cdots, p_k \in \overline{\mathcal{O}}(t)$, and they are pairwise disjoint;
- $t[x_1 \cdots, x_k]_{\{p_1,\cdots,p_k\}}$ contains only one occurrence of a function symbol, where x_1, $\cdots$, x_k are new variables; and
- each $t|_{p_i}$ $(1 \leq i \leq k)$ has the form $f\ s_1 \cdots s_n$ with f a function symbol.

Term $t[x_1 \cdots, x_k]_{\{p_1,\cdots,p_k\}}$ is obtained from t by replacing each subterm at position p_i with x_i simultaneously, and will be denoted as $[t]$. For example $x +\!\!+ (y +\!\!+ z)$ can be abstracted as $x +\!\!+ (y +\!\!+ z)|_{\{2\}}$ and $[x +\!\!+ (y +\!\!+ z)] = x +\!\!+ x_1$.

Lemma 2.3 The combinator abstraction of any term t is unique up to variable renaming. $\square$

We write $t \rightsquigarrow^* t'$ if there is a **structural narrowing derivation**

$$t = t_1, [t_1] \rightsquigarrow_{l_1=r_1} t_2, [t_2] \rightsquigarrow_{l_2=r_2} t_3, \cdots, [t_{n-1}] \rightsquigarrow_{l_{n-1}=r_{n-1}} t_n = t'.$$

The following is an example:

$$\begin{array}{lll} x * y & \rightsquigarrow_{(succ\ x)*y=y+(x*y)} & y_1 + (x_1 * y_1) \quad (1) \\ y_1 + z & \rightsquigarrow_{(succ\ x)+y=succ(x+y)} & succ(x_2 + y_2) \quad (2) \end{array}$$

In (1), the equation $(succ\ x)*y = y+(x*y)$ is renamed to $(succ\ x_1)*y_1 = y_1+(x_1*y_1)$ and the substitution is $\{x \mapsto succ\ x_1,\ y \mapsto y_1\}$; in (2) the equation $(succ\ x)+y = succ(x+y)$ is renamed to $(succ\ x_2)+y_2 = succ(x_2+y_2)$ and the substitution is $\{y_1 \mapsto succ\ x_2,\ z \mapsto y_2\}$. $y_1 + (x_1 * y_1)$ is abstracted as $y_1 + (x_1 * y_1)|_{\{2\}}$ and $[y_1 + (x_1 * y_1)] = y_1 + z$.

The system $\mathcal{TS}^1$

Let $\Sigma = \langle \mathcal{S}, \mathcal{C}, \mathcal{F} \rangle$, $\mathcal{SP} = \langle \Sigma, \mathcal{E} \rangle$, and Γ a matching problem of the specification $\mathcal{SP}$. In the sequel, we write $l = r \in \mathcal{E}$ to denote a renaming of an equation in $\mathcal{E}$, whose variables do not occur in other related terms, otherwise we can rename it. If we write $f\ t_1 \cdots t_n$, we mean that each t_i $(1 \leq i \leq n)$ does not contain any matching combinator, otherwise it is presented in its abstracted form. It is assumed that every term is automatically normalized with respect to $\mathcal{E}$.

The system $\mathcal{TS}^1$ for solving matching problems of constructor-based specifications consists of the following six rules.

(Remove) removes a trivial matching pair:

$$\{(t, t)\} \cup \Gamma \Longrightarrow \Gamma.$$

(Deletion) deletes a matching pair playing no role in solutions:

$$\{(t, x)\} \cup \Gamma \Longrightarrow \Gamma,$$

if $x \notin \mathcal{V}(\Gamma)$.

(Variable) inserts a partial solution for a variable:

$$\{(x,t)\} \cup \Gamma \Longrightarrow \{(x,t)\} \cup (\{x \mapsto t\}\Gamma), \qquad \text{if } x \in \mathcal{V}(\Gamma).$$

(Decomposition) breaks a matching pair down into simpler ones:

$$\{(f\ t_1 \cdots\ t_n, f\ s_1 \cdots s_n)\} \cup \Gamma \Longrightarrow \{(t_1,s_1),\cdots,(t_n,s_n)\} \cup \Gamma,$$

$f \in \mathcal{C} \cup \mathcal{F}$.

(Abstraction) abstracts a matching pair:

$$\{(t|_{\{p_1 \cdots, p_k\}}, s)\} \cup \Gamma \Longrightarrow \{(t[x_1,\cdots,x_k]_{\{p_1,\cdots,p_k\}}, s), (t|_{p_1}, x_1), \cdots, (t|_{p_k}, x_k)\} \cup \Gamma,$$

where $\{x_1,\cdots,x_k\} \cap (\mathcal{V}(t|_{p_1},\cdots,t|_{p_k}) \cup \mathcal{V}(s) \cup \mathcal{V}(\Gamma)) = \emptyset$.

(Narrowing) realizes structural narrowing of a pair:

$$\{(f\ t_1 \cdots t_n, s)\} \cup \Gamma \Longrightarrow \Gamma_\sigma \cup \{(\sigma r, s)\} \cup (\sigma\Gamma),$$

where $l = r \in \mathcal{E}$, $\sigma = mgu(f\ t_1 \cdots t_n, l)$, and $\mathcal{I}(\sigma) \cap (\mathcal{V}(s) \cup \mathcal{V}(\Gamma)) = \emptyset$. Remember that Γ_σ is the solved matching problem formed from σ.

As usual, $\Longrightarrow^*$ denotes the reflexive-transitive closure of $\Longrightarrow$. (Deletion), (Abstraction) and (Narrowing) make the transformation system different from the system solving unification problems. The rule (Narrowing) uses structural narrowing to search for possible solutions. Let $\{(x+y, succ(succ\ 0))\}$ be a matching problem, $(succ\ x) + y = succ(x+y)$ the equation to be applied. At first, the equation is renamed to $(succ\ x_1) + y_1 = succ(x_1 + y_1)$. The most general unifier of $x + y$ and $(succ\ x_1) + y_1$ is $\{x \mapsto succ\ x_1, y \mapsto y_1\}$. So,

$$\begin{aligned}&\{(x+y, succ(succ\ 0))\}\\ &\Longrightarrow_{(\text{Narrowing})} \{(x, succ\ x_1), (y, y_1), (succ(x_1+y_1), succ(succ\ 0))\}.\end{aligned}$$

To solve this matching problem, we may have the following sequence of transformations:

$$\begin{aligned}&\{(x+y, succ(succ\ 0))\}\\ &\Longrightarrow_{(\text{Narrowing})} \{(x, succ\ x_1), (y, y_1), (succ(x_1+y_1), succ(succ\ 0))\}\\ &\Longrightarrow_{(\text{Decomposition})} \{(x, succ\ x_1), (y, y_1), (x_1+y_1, succ\ 0)\}\\ &\Longrightarrow_{(\text{Narrowing})} \{(x, succ\ 0), (y, y_2), (x_1, 0), (y_1, y_2), (y_2, succ\ 0)\}\\ &\Longrightarrow_{(\text{Variable})} \{(x, succ\ 0), (y, succ\ 0), (x_1, 0), (y_1, succ\ 0), (y_2, succ\ 0)\}.\end{aligned}$$

If Γ is transformed to a solved form Γ', then $\sigma_{\Gamma'}[\mathcal{V}(\Gamma)]$ is a match of Γ by $\mathcal{TS}^1$. In our example, it is $\{x \mapsto succ\ 0,\ y \mapsto succ\ 0\}$.

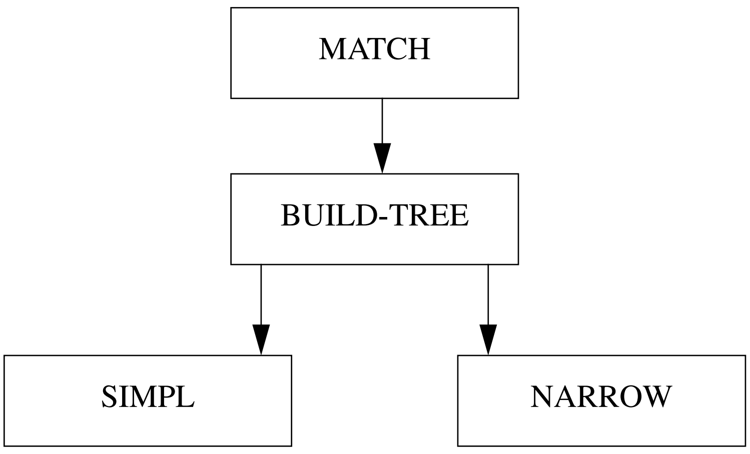

Figure 1: The structure of $\mathcal{MA}^1$

2.2.

The rule (Deletion) throws away matching pairs playing no role in final solutions that could be produced by (Abstraction) and (Narrowing). This can be seen from the following example:

$$\begin{aligned}
&\{(len((x+y) :: xs), succ(succ\ 0))\}\\
&\Longrightarrow_{\text{(Abstraction)}} \{(len(z :: xs), succ(succ\ 0)), (x+y, z)\}\\
&\longrightarrow \{(succ(len\ xs), succ(succ\ 0)), (x+y, z)\}\\
&\Longrightarrow_{\text{(Decomposition)}} \{(len\ xs, succ\ 0), (x+y, z)\}\\
&\Longrightarrow_{\text{(Deletion)}} \{(len\ xs, succ\ 0)\}\\
&\Longrightarrow_{\text{(Narrowing)}} \{(xs, x_1 :: xs_1), (succ(len\ xs_1), succ\ 0)\}\\
&\Longrightarrow_{\text{(Decomposition)}} \{(xs, x_1 :: xs_1), (len\ xs_1, 0)\}\\
&\Longrightarrow_{\text{(Narrowing)}} \{(xs, x_1 :: nil), (xs_1, nil), (0, 0)\}\\
&\Longrightarrow_{\text{(Remove)}} \{(xs, x_1 :: nil), (xs_1, nil)\}
\end{aligned}$$

The match is $\{xs \mapsto x_2 :: nil\}$. In $\{(len\ xs,\ succ\ 0), (x+y, z)\}$, z does not occur anywhere else, so the pair $(x+y, z)$ can be deleted.

2.2.2 The Algorithm $\mathcal{MA}^1$

The matching algorithm $\mathcal{MA}^1$ contains four modules MATCH, BUILD-TREE, SIMPL and NARROW. Figure 1 shows its structure, where directed arrows present the dependency relation.

The central idea of the algorithm is to construct a **matching tree** (BUILD-TREE) for any given matching problem Γ using SIMPL and NARROW. Each node is labeled by a matching problem; the root is Γ. If there is a node Γ' that is not in solved form, simplify it first using SIMPL, i.e., apply the rules (Remove), (Deletion), (Decomposition), (Variable) and (Abstraction) exhaustively. Then choose a pair to apply all possible structural narrowing (Narrowing) and get a set of new matching problems by NARROW.

For each of them draw an edge from Γ' to it, until all nodes are in solved form, or no rules can be applied. The former are called **success nodes**, the latter **failure nodes**. All success nodes form the set of matches of Γ.

In the following we describe SIMPL and NARROW abstractly, where some abbreviations are used, e.g., *Decomposition?*(Γ) means whether rule (Decomposition) is applicable to Γ, i.e., whether there is a pair $(f\ s_1 \cdots s_n, f\ t_1 \cdots t_n) \in \Gamma$; *Decomposition*($\Gamma$) means applying the (Decomposition) transformation to Γ.

$\mathsf{SIMPL}(\Gamma)$:
if $(c_1\ t_1 \cdots t_n, c_2\ s_1 \cdots s_n) \in \Gamma \wedge c_1 \neq c_2$ then *failure*
elseif $\Gamma = \{(t,t)\} \cup \Gamma'$ then $\mathsf{SIMPL}(\Gamma')$
elseif *Deletion?*(Γ) then $\mathsf{SIMPL}(\mathit{Deletion}(\Gamma))$
elseif *Decomposition?*(Γ) then $\mathsf{SIMPL}(\mathit{Decomposition}(\Gamma))$
elseif *Variable?*(Γ) then $\mathsf{SIMPL}(\mathit{Variable}(\Gamma))$
elseif *Abstraction?*(Γ) then $\mathsf{SIMPL}(\mathit{Abstraction}(\Gamma))$
else Γ
end SIMPL.

$\mathsf{NARROW}(\{(s,t)\} \cup \Gamma', \mathcal{E})$:
if $\mathcal{E} = \mathit{empty}$ then $\{\}$
if $\mathcal{E} = \{l = r\} \cup \mathcal{E}'$ then
  if *Unifiable?* (s, l)
    then $\{\mathit{Narrowing}(\{(s,t)\} \cup \Gamma', l = r)\} \cup \mathsf{NARROW}(\{(s,t)\} \cup \Gamma', \mathcal{E}')$
    else $\mathsf{NARROW}(\{(s,t)\} \cup \Gamma', \mathcal{E}')$
end NARROW.

$\mathit{Narrowing}(\{(s,t)\} \cup \Gamma', l = r)$ means applying the equation $l = r$ to narrow the pair (s,t) in the matching problem $\{(s,t)\} \cup \Gamma'$. The result is a new matching problem. Figure 2 shows a matching tree for the matching problem $\{(x + y, \mathit{succ}(\mathit{succ}(\mathit{succ}\ 0)))\}$. The simplification steps are omitted, the equations used by (Narrowing) are given next to edges. The matches produced are:

$$\{x \mapsto 0, y \mapsto \mathit{succ}(\mathit{succ}(\mathit{succ}\ 0))\}$$

$$\{x \mapsto \mathit{succ}\ 0, y \mapsto \mathit{succ}(\mathit{succ}\ 0)\}$$

$$\{x \mapsto \mathit{succ}(\mathit{succ}\ 0), y \mapsto \mathit{succ}\ 0\}$$

$$\{x \mapsto \mathit{succ}(\mathit{succ}(\mathit{succ}\ 0)), y \mapsto 0\}$$

$\{ (x+y, succ(succ(succ\ 0))) \}$

$0+y_1 = y_1$

$\{ (x, 0), (y, succ(succ(succ\ 0))), (y_1, succ(succ(succ\ 0))) \}$

$(succ\ x_1)+y_1 = succ(x_1+y_1)$

$\{ (x, succ\ x_1), (y, y_1), (x_1+y_1, succ(succ\ 0)) \}$

$0+y_2 = y_2$

$\{ (x, succ\ 0), (y, succ(succ\ 0)), (x_1, 0), (y_1, succ(succ\ 0)), (y_2, succ(succ\ 0)) \}$

$(succ\ x_2)+y_2 = succ(x_2+y_2)$

$\{ (x, succ(succ\ x_2)), (y, y_2), (x_1, succ\ x_2), (y_1, y_2), (x_2+y_2, succ\ 0) \}$

$0+y_3 = y_3$

$\{ (x, succ(succ\ 0)), (y, succ\ 0), (x_1, succ\ 0), (y_1, succ\ 0), (x_2, 0), (y_2, succ\ 0), (y_3, succ\ 0) \}$

$(succ\ x_3)+y_3 = succ(x_3+y_3)$

$\{ (x, succ(succ(succ\ x_3))), (y, y_3), (x_1, succ(succ\ x_3)), (y_1, y_3), (x_2, succ\ x_3), (y_2, y_3), (x_3+y_3, 0) \}$

$0+y_4 = y_4$

$\{ (x, succ(succ(succ\ 0))), (y, 0), (x_1, succ(succ\ 0)), (y_1, 0), (x_2, succ\ 0), (y_2, 0), (x_3, 0), (y_3, 0), (y_4, 0) \}$

$(succ\ x_4)+y_4 = succ(x_4+y_4)$

failure

Figure 2: A matching tree

2.3 Improvements of $\mathcal{M}A^1$

A direct implementation of the matching algorithm $\mathcal{M}A^1$ in SML [33] shows that it can become inefficient even for some simple matching problems. The following specification *Tree* defines the function *height* on binary trees, where *max* is the maximum function defined in *Nat*.

$$
\begin{array}{lll}
\mathit{Tree} & = \mathit{Nat} + & \\
 & \textbf{sorts}: & \mathit{tree} \\
 & \textbf{cons}: & \mathit{empty} \ : \ \mathit{tree} \\
 & & \mathit{node} \ : \ \mathit{tree} \to \mathit{tree} \to \mathit{tree} \\
 & \textbf{funs}: & \mathit{height} \ : \ \mathit{tree} \to \mathit{nat} \\
 & \textbf{eqns}: & \forall\, x, y \in \mathit{tree}: \\
 & & \mathit{height}\ \mathit{empty} = \ 0 \\
 & & \mathit{height}(\mathit{node}\ x\ y) = \mathit{succ}(\mathit{max}\ (\mathit{height}\ x)\ (\mathit{height}\ y))
\end{array}
$$

The direct implementation takes more than half an hour to compute all the possible trees of height 4, i.e., to solve the matching pair:

$$(\mathit{height}(\mathit{node}\ t_1\ t_2), \mathit{succ}(\mathit{succ}(\mathit{succ}(\mathit{succ}\ 0)))).$$

Let $\#(n)$ denote the number of trees of height n. Formally, we can use the following recursive definition to show that there are 651 trees of height 4.

$$
\begin{array}{lll}
\#(0) & = & 1 \\
\#(n+1) & = & 2 * (\sum_{1 \le i \le n-1} \#(i) * \#(n)) + \#(n) * \#(n)
\end{array}
$$

There is only one tree of height 0, i.e., *empty*. A tree of height $n+1$ has the form *node* t_1 t_2, where the height of t_1 or t_2 should be n, and the height of the other one can be less than n; or both t_1 and t_2 are of height n.

For practical use, additional techniques to improve the algorithm are indispensable. The techniques we applied are:

- Dependency analysis — independent matching pairs will be solved first. It helps to reduce the sizes of matching trees, and offers a direct way to analyse the algorithm as well.

- Partial sharing analysis — a matching pair will never be solved more than once. With this technique, redundant matching steps can be avoided, but it can sometimes be expensive.

- Commutative matching — for a matching pair with a commutative function, only some of its solutions will be computed, the rest can be obtained by using the commutative property.

After applying these techniques, our algorithm has obviously been improved: all 651 trees of height 4 are produced in 3 minutes.

2.3.1 Dependency Analysis

Dependency analysis is the most important technique we applied to improve the efficiency of our matching algorithm. It will be used to prove the correctness of the algorithm as well, see section 2.4. Dependency analysis is motivated by the partial ordering for first-order unification problems from Martelli and Montanari [52]. One of the reasons to make $\mathcal{MA}^1$ inefficient is the random selection of matching pairs to apply (Narrowing). In particular, the uses of (Narrowing) and (Abstraction) may transform ground matching problems into non-ground problems. See the following examples:

$$\begin{array}{ll} \{(x_1 +\!\!+ (x_2 +\!\!+ x_3), 3 :: (2 :: nil))\} & \\ \Longrightarrow_{\text{(Abstraction)}} \{(x_1 +\!\!+ z, 3 :: (2 :: nil)), (x_2 +\!\!+ x_3, z)\} & (1) \end{array}$$

$$\begin{array}{ll} \{(x_1 +\!\!+ z, 3 :: (2 :: nil))\} & \\ \Longrightarrow_{\text{(Narrowing)}} \{(x_1, x :: xs), (z, ys), (x :: (xs +\!\!+ ys), 3 :: (2 :: nil))\} & (2) \end{array}$$

The right hand sides of both (1) and (2) are not ground matching problems, they contain newly introduced variables z, ys, x and xs. It is not reasonable to say that there exists a substitution σ, such that $\sigma(x_2 +\!\!+ x_3) =_{\mathcal{E}} z$ with z a variable occurring also in the matching pair $(x_1 +\!\!+ z, 3 :: (2 :: nil))$. Finally, solving $(x_2 +\!\!+ x_3, z)$ first without knowing z could be very inefficient through many useless narrowing steps. Instead, the pairs whose right-hand sides contain no variables that are dependent upon other pairs, such as $(x_1 +\!\!+ z, 3 :: (2 :: nil))$, $(x_1, x : xs)$ and $(x :: (xs +\!\!+ ys), 3 :: (2 :: nil))$, should be considered first.

Definition 2.17 (Dependency and solvable matching pairs) A matching pair (s, t) is called **dependent on** pair (s', t'), if $\mathcal{V}(t) \cap \mathcal{V}(s') \neq \emptyset$, denoted as $(s, t) \prec (s', t')$. A matching pair $(s, t) \in \Gamma$ is **solvable**, if it is not in solved form and there is no (s', t') $\in \Gamma$ such that $(s, t) \prec (s', t')$.

The algorithm $\mathcal{MA}_d^1$

Now we give a new version of the matching algorithm.

Definition 2.18 (Representation) Let Γ be a matching problem, the representation of Γ is denoted as $\Gamma \stackrel{rep}{=} \Gamma_0 \ \& \ \Gamma_1 \ \& \ \Gamma_2$ such that,

- $\Gamma_0 \cup \Gamma_1 \cup \Gamma_2 = \Gamma$,
- Γ_0 contains all solved pairs, Γ_1 all solvable pairs, and Γ_2 the rest.

If Γ is a ground matching problem or a solved matching problem, then it will be represented as $\{\} \ \& \ \Gamma \ \& \ \{\}$ or $\Gamma \ \& \ \{\} \ \& \ \{\}$ respectively. With this representation, the rules (Variable), (Abstraction) and (Narrowing) in $\mathcal{TS}_d^1$ are as follows:

(Variable) inserts a partial solution for a variable:

$$\Gamma_0 \ \& \ \{(x, t)\} \cup \Gamma_1 \ \& \ \Gamma_2 \Longrightarrow \{(x, t)\} \cup (\sigma\Gamma_0) \ \& \ (\sigma\Gamma_1) \cup (\sigma\Gamma_{2,1}) \ \& \ (\sigma\Gamma_{2,2}),$$

where $\sigma = \{x \mapsto t\}$ and $\Gamma_{2,1} \cup \Gamma_{2,2} = \Gamma_2$.

(Abstraction) abstracts a matching pair:

$$\begin{array}{l}\Gamma_0 \ \& \ \{(t|_{\{p_1 \cdots p_k\}}, s)\} \cup \Gamma_1 \ \& \ \Gamma_2 \\ \Longrightarrow \Gamma_0 \ \& \ \{(t[x_1, \cdots, x_k]_{\{p_1, \cdots, p_n\}}, s)\} \cup \Gamma_1 \ \& \ \{(t|_{p_1}, x_1), \cdots, (t|_{p_k}, x_k)\} \cup \Gamma_2,\end{array}$$

where x_1, $\cdots$, x_k are new variables that do not occur anywhere else.

(Narrowing) realizes structural narrowing of a pair:

$$\Gamma_0 \ \& \ \{(f\ t_1 \cdots\ t_n, t)\} \cup \Gamma_1 \ \& \ \Gamma_2 \Longrightarrow \Gamma_\sigma \cup (\sigma\Gamma_0) \ \& \ \{(\sigma r, t)\} \cup (\sigma\Gamma_1) \cup (\sigma\Gamma_{2,1}) \ \& \ (\sigma\Gamma_{2,2}),$$

where $l = r \in \mathcal{E}$, $\sigma = mgu(f\ t_1 \cdots t_n, l)$, $\Gamma_{2,1} \cup \Gamma_{2,2} = \Gamma_2$, and the variables in $\mathcal{I}(\sigma)$ do not occur anywhere else.

The rules (Remove), (Deletion) and (Decomposition) are not changed. That is, they can be applied to any matching pair in a matching problem. (Variable), (Abstraction) and (Narrowing) can only be applied to solvable matching pairs, and they may change Γ_2.

Obviously, if

$$\Gamma \stackrel{rep}{=} \Gamma_0 \ \& \ \Gamma_1 \ \& \ \Gamma_2 \Longrightarrow_{\mathcal{TS}^1_d} \Gamma'_0 \ \& \ \Gamma'_1 \ \& \ \Gamma'_2 \stackrel{rep}{=} \Gamma',$$

then there exists a transformation: $\Gamma \Longrightarrow_{\mathcal{TS}^1} \Gamma'$. In section 2.4 we will show that if $\Gamma \Longrightarrow^*_{\mathcal{TS}^1} \Gamma'$, where Γ' is in solved form, then there exists $\Gamma \Longrightarrow^*_{\mathcal{TS}^1_d} \Gamma''$, such that $\sigma_{\Gamma''} \leq_{[\mathcal{V}(\Gamma)]} \sigma_{\Gamma'}$.

A solvable matching pair (t, s) in a matching problem can be solved independently, because the variables in s will not be substituted further. So we can treat a solvable matching pair as a new matching problem, solve it using the matching algorithm and get a set of matches. Finally, the matches will be substituted into the original matching problem individually.

The following theorem ensures that the definition of solvable matching pairs is reasonable. The left linearity of constructor-based specifications is important to prove it.

Theorem 2.1 Let Γ be a solvable ground matching problem and $\Gamma \Longrightarrow^*_{\mathcal{TS}^1_d} \Gamma'$. Γ' contains at least one solvable matching pair as long as it is not in solved form.

At first we present two lemmas.

Lemma 2.4 Let t be any term and l a linear term, i.e. every variable in l occurs only once, such that $\mathcal{V}(t) \cap \mathcal{V}(l) = \emptyset$. If $\sigma = mgu(t, l)$, then

- σx is linear for any $x \in \mathcal{V}(t)$;
- $\mathcal{V}(\sigma x_1) \cap \mathcal{V}(\sigma x_2) = \emptyset$ for any x_1 and $x_2 \in \mathcal{V}(t)$.

Proof. By structural induction over l:

- If l is a variable, then $\sigma = \{l \mapsto t\}$, the lemma is obviously true.
- If l is a constant, then $\mathcal{I}(\sigma) = \emptyset$.

- If l has the form $f\ l_1\ l_2$ (for simplicity), f may be a constructor or a function symbol. Suppose t has the form $f\ t_1\ t_2$, otherwise they are not unifiable. Let $\sigma_1 = mgu(t_1, l_1)$ and $\sigma_2 = mgu(\sigma_1\ t_2, \sigma_1\ l_2)$, then $\sigma = \sigma_2 \circ \sigma_1$. Because l is linear, $\mathcal{V}(t) \cap \mathcal{V}(l) = \emptyset$, and the fact that $\mathcal{D}(\sigma_1) \cup \mathcal{I}(\sigma_1) = \mathcal{V}(t_1) \cup \mathcal{V}(l_1)$ (see [54]), we have $\mathcal{D}(\sigma_1) \cap \mathcal{V}(l_2) = \emptyset$ and $\sigma_1\ l_2 = l_2$. Using the induction hypotheses, it is easy to show that σx is linear for any $x \in \mathcal{V}(t)$. Now we prove that $\mathcal{V}(\sigma x_1) \cap \mathcal{V}(\sigma x_2) = \emptyset$ for any x_1 and $x_2 \in \mathcal{V}(t)$. The case of x_1 and $x_2 \in \mathcal{V}(t_1)$ or x_1 and $x_2 \in \mathcal{V}(t_2)$ can be proved directly by using induction hypotheses. If $x_1 \in \mathcal{V}(t_1)$ and $x_2 \in \mathcal{V}(t_2)$, we can show that $x_2 \notin \mathcal{V}(\sigma_1 x_1)$, using the fact that $\mathcal{D}(\sigma_1) \cup \mathcal{I}(\sigma_1) = \mathcal{V}(t_1) \cup \mathcal{V}(l_1)$, so $\mathcal{V}(\sigma x_1) \cap \mathcal{V}(\sigma x_2) = \emptyset$. □

Using this lemma and the fact that each constructor-based specification is left linear, the following lemma can be proved by induction over transformations.

Lemma 2.5 Let Γ be a ground matching problem and $\Gamma \Longrightarrow^*_{\mathcal{TS}^1_d} \Gamma'$,

- if $(t, s) \in \Gamma$, then s is linear;
- if $(t_1, s_1), (t_2, s_2) \in \Gamma'$, then $\mathcal{V}(s_1) \cap \mathcal{V}(s_2) = \emptyset$. □

Now theorem 2.1 can be proved by the following lemma.

Lemma 2.6 If Γ is a ground matching problem and $\Gamma \Longrightarrow^*_{\mathcal{TS}^1_d} \Gamma' \stackrel{rep}{=} \Gamma'_0\ \&\ \Gamma'_1\ \&\ \Gamma'_2$, then the dependence relation $\prec$ is well-founded on Γ'_2.

Proof. By induction on transformations:

The lemma is true for Γ, since Γ is a ground matching problem. Let $\Gamma_0\ \&\ \Gamma_1\ \&\ \Gamma_2 \Longrightarrow_{\mathcal{TS}^1_d} \Gamma'_0\ \&\ \Gamma'_1\ \&\ \Gamma'_2$, and $\prec$ is well-founded on Γ_2, then we should show that $\prec$ is well-founded on Γ'_2. The only rules that need to be considered are (Variable), (Abstraction) and (Narrowing).

- (Variable). Let $\sigma = \{x \mapsto t\}$, $\Gamma_{2,1} \cup \Gamma_{2,2} = \Gamma_2$ and (s_1, s_2), $(s'_1, s'_2) \in \Gamma_{2,2}$. We show that if $(s_1, s_2) \not\prec (s'_1, s'_2)$, i.e., $\mathcal{V}(s_2) \cap \mathcal{V}(s'_1) = \emptyset$, then $(\sigma s_1, \sigma s_2) \not\prec (\sigma s'_1, \sigma s'_2)$. There are three cases to be considered:
  - If $x \notin \mathcal{V}(s_2)$ and $x \notin \mathcal{V}(s'_1)$, then $\mathcal{V}(\sigma s_2) \cap \mathcal{V}(\sigma s'_1) = \emptyset$.
  - If $x \in \mathcal{V}(s_2)$, but $x \notin \mathcal{V}(s'_1)$, because $\mathcal{V}(t) \cap \mathcal{V}(s'_1) = \emptyset$ ((x, t) is solvable), so $\mathcal{V}(\sigma s_2) \cap \mathcal{V}(s'_1) = \emptyset$.
  - If $x \notin \mathcal{V}(s_2)$, but $x \in \mathcal{V}(s'_1)$, because $\mathcal{V}(t) \cap \mathcal{V}(s_2) = \emptyset$ (lemma 2.5), so $\mathcal{V}(s_2) \cap \mathcal{V}(\sigma s'_1) = \emptyset$.

  That is, $\mathcal{V}(\sigma s_2) \cap \mathcal{V}(\sigma s'_1) = \emptyset$.
- (Abstraction). Let $(s_1, s_2) \in \Gamma_2$, then any new variable $x_i \notin \mathcal{V}(s_1)$. So $(t|_{p_i}, x_i) \not\prec (s_1, s_2)$.
- (Narrowing). The proof is similar to the case of (Variable). □

Without left linearity, this lemma does not always hold. The following is a counterexample. Let f and g be two function symbols. One of the equations associated with f is $f\ (c\ x)\ x = a$, where c and a are constructors. We have the matching problem $\{(f\ (g\ y)\ y,\ a)\}$ and the following transformations:

$$
\begin{aligned}
&\{\}\ \&\ \{(f\ (g\ y)\ y, a)\}\ \&\ \{\}\\
&\Longrightarrow_{(\text{Abstraction})}\ \{\}\ \&\ \{(f\ y_1\ y, a)\}\ \&\ \{(g\ y, y_1)\}\\
&\Longrightarrow_{(\text{Narrowing})}\ \{(y_1, c\ x), (y, x)\}\ \&\ \{(a, a)\}\ \&\ \{(g\ x, c\ x)\}\\
&\Longrightarrow_{(\text{Remove})}\ \{(y_1, c\ x), (y, x)\}\ \&\ \{\}\ \&\ \{(g\ x, c\ x)\}.
\end{aligned}
$$

Although the problem is not in solved form, there is no solvable pair.

The algorithm $\mathcal{MA}_d^1$ contains the main module MATCH, which produces a set of matches for a given matching problem. MATCH uses BUILD-TREE which depends on SIMPL and SOLVE, and the new module SOLVE uses NARROW and MATCH to solve a solvable matching pair. See figure 3 about its structure.

The central idea of the algorithm $\mathcal{MA}_d^1$ is also to construct a matching tree for any given matching problem Γ. The root is Γ. If there is a simplified node Γ' in the matching tree, which is not in solved form, then choose a solvable pair from it and solve it independently using SOLVE. For each match σ of the pair draw an edge from Γ' to $\sigma\Gamma'$. To solve a solvable pair, at first apply all possible structural narrowing (Narrowing) to it and get a set of new matching problems by NARROW, then use MATCH to solve each of these problems. Here we give a brief description of SOLVE.

SOLVE$(t, s, \mathcal{E})$:
let $NS =$ NARROW$(\{(t, s)\}, \mathcal{E})$
in map(MATCH, NS)
end SOLVE.

(t, s) is the solvable matching pair chosen; NS is the set of new matching problems obtained by applying all possible structural narrowing to the pair. The function map applies the procedure MATCH to each new matching problem.

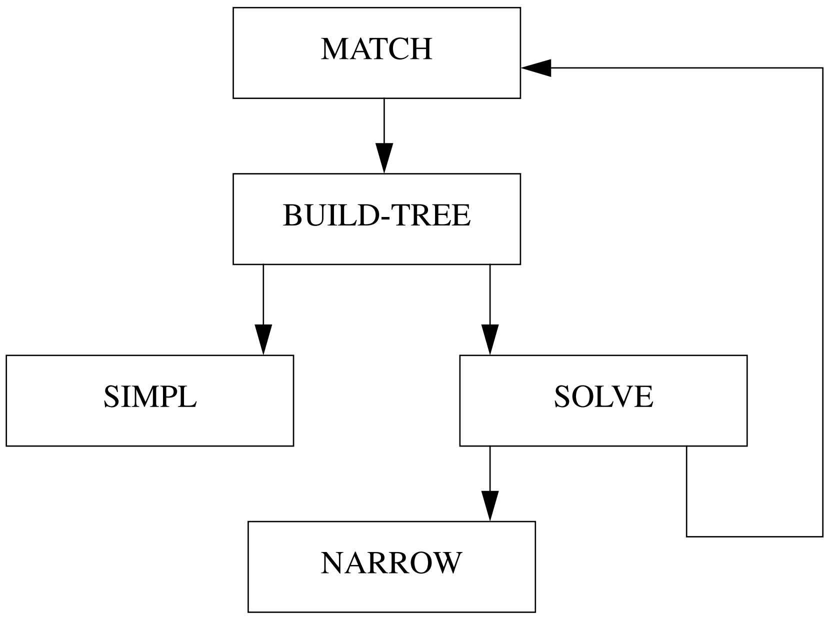

Figure 3: The structure of $\mathcal{MA}_d^1$

2.3.

Figure 4 shows a matching tree produced by $\mathcal{MA}_d^1$ for the matching problem $\{(height\ (node\ t_1\ t_2), succ(succ\ 0))\}$. For easy comparison, we also give a matching tree produced by $\mathcal{MA}^1$ for the same problem in figure 5. Although we omitted many parts in figure 5, the matching tree is too large to be put on one page.

In figure 5, all narrowing steps are recorded in the tree, and variables like x and y are substituted repeatedly. In figure 4, only solutions of solvable pairs are recorded, so such a tree is never higher than the number of the matching pairs to be solved. In our example it is 3. In addition, variables in the original problem like x and y will be substituted only once.

2.3.2 Partial Sharing Analysis

In this section partial sharing analysis will be discussed, which 'remembers' the matching pairs that are already solved. However, we will see that it can sometimes be expensive.

Definition 2.19 (Similar matching pairs) Two matching pairs (t, s) and (t', s') are called similar, if there is a renaming σ such that $\sigma t = t'$ and $\sigma s = s'$.

In a matching tree there are often some similar matching pairs. For instance, (*height* t_1, *succ* 0) and (*height* t_2, *succ* 0) are similar, and each of them occurs two times, see figure 4. Usually, all of these matching pairs will be solved by SOLVE, because algorithm $\mathcal{MA}_d^1$ does not "remember" the matching pairs which are already solved, i.e., which have already been transformed into their solved forms (matches). To avoid redundant

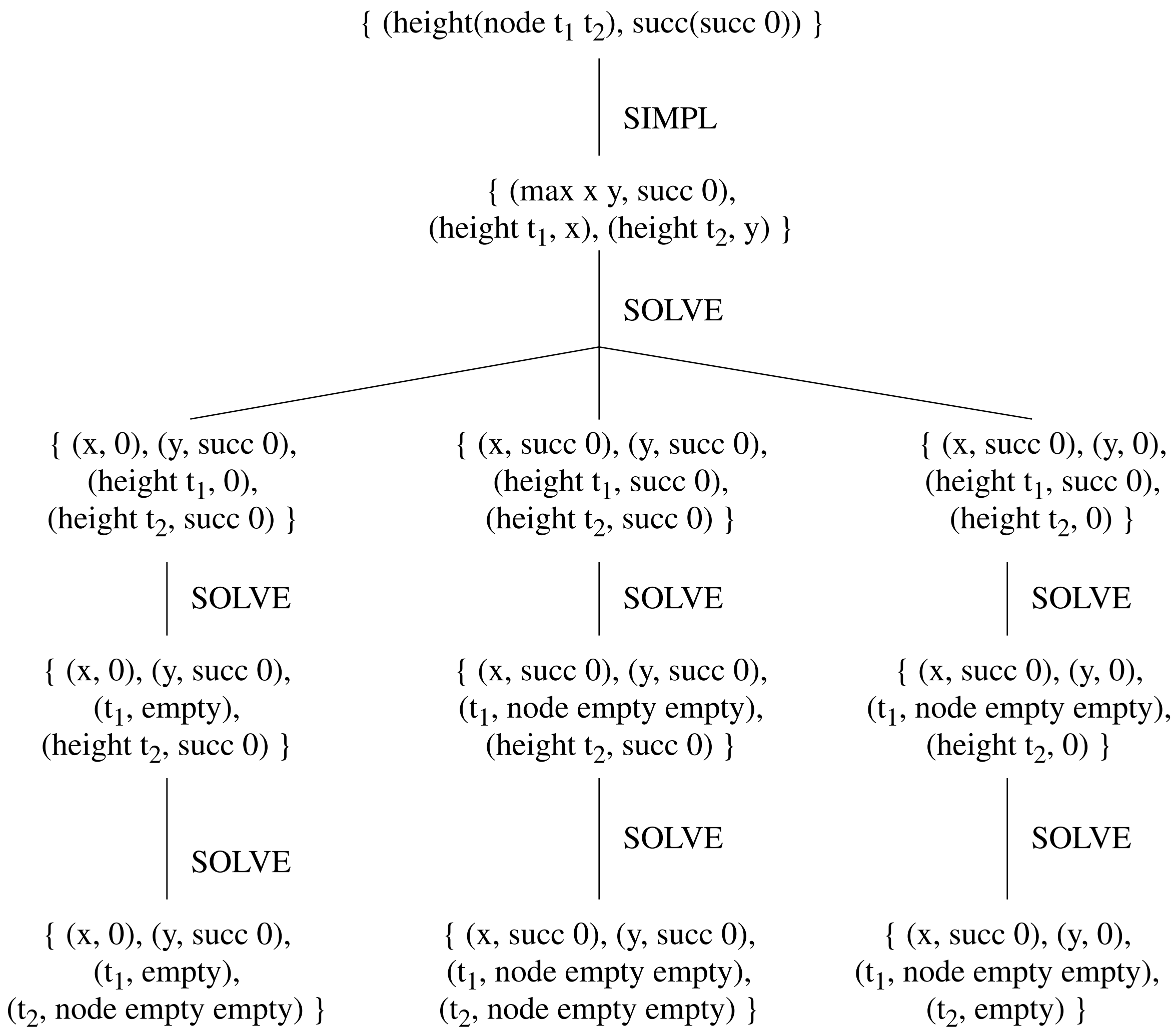

Figure 4: A matching tree produced by MA_d^1

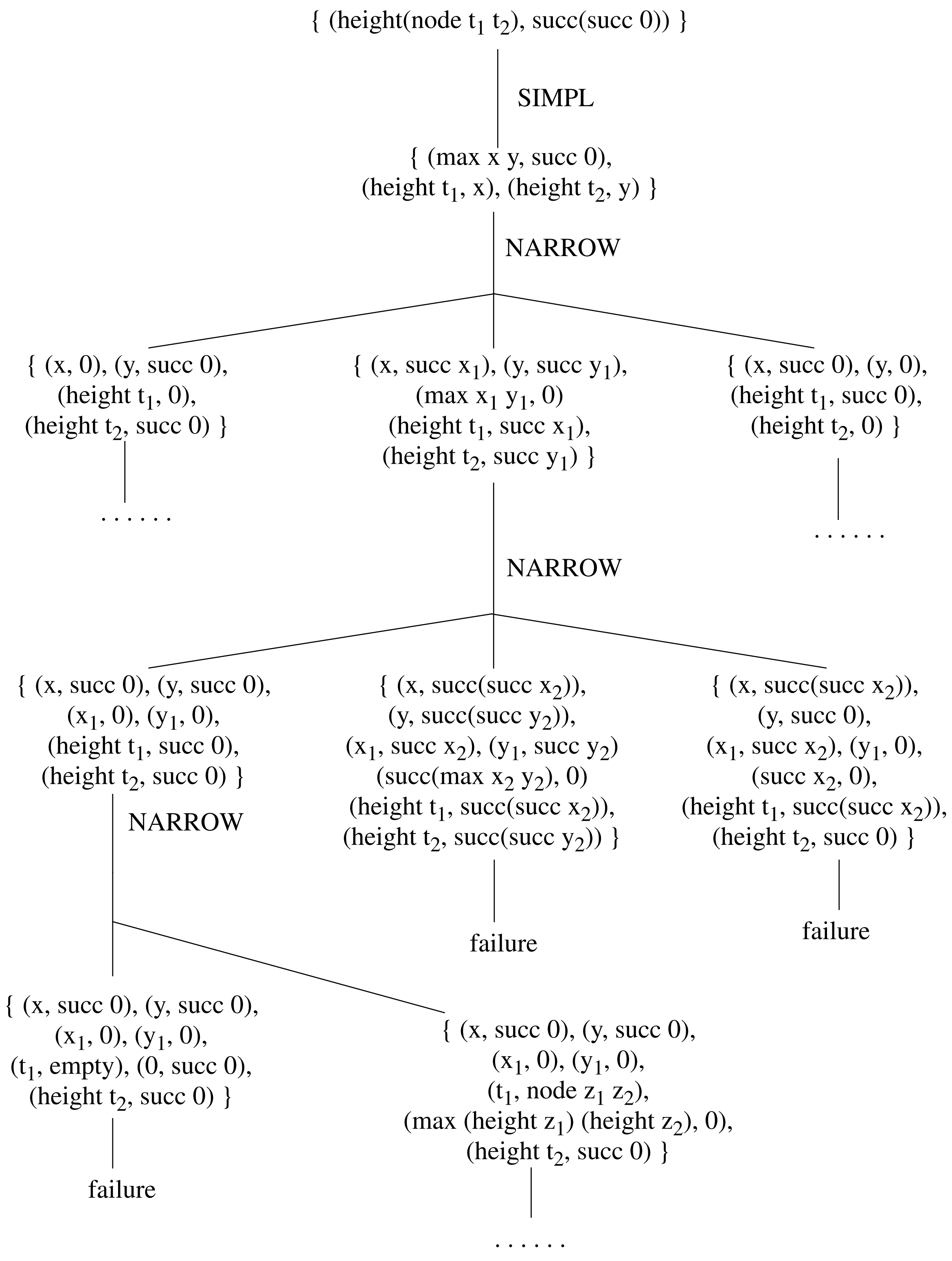

Figure 5: Part of a matching tree produced by $\mathcal{MA}^1$

solutions of similar matching pairs, we define a **global solution list** for each given matching problem, in which solved pairs with their matches will be recorded.

Algorithm $\mathcal{M}A_p^1$ also contains a main procedure MATCH, which uses BUILD-TREE, SIMPL, and SOLVE. After choosing a solvable matching pair from a matching problem, we check first whether a similar pair exists in the global solution list or not. If it is already solved, we only need to take its matches from the solution list; otherwise, it will be solved by SOLVE and will be added to the solution list. In the above example, the solution list contains solutions of $(max\ x\ y,\ succ\ 0)$, $(max\ x\ y,\ 0)$, $(height\ t,\ 0)$, and $(height\ t,\ succ\ 0)$. The number of matching pairs to be solved is now 4 instead of more than 12 by using $\mathcal{M}A_d^1$; for instance, solving $(height\ t_1,\ succ\ 0)$ by $\mathcal{M}A_d^1$ contains again solving $(max\ x\ y,\ 0)$, $(height\ t,\ 0)$, which can be seen from the following transformations:

$$
\begin{array}{ll}
 & \{(height\ t_1, succ\ 0)\} \\
\Longrightarrow_{(\text{Narrowing})} & \{(t_1, node\ t_1'\ t_2'), (succ(max\ (height\ t_1')\ (height\ t_2')), succ\ 0)\} \\
\Longrightarrow_{(\text{Decomposition})} & \{(t_1, node\ t_1'\ t_2'), (max\ (height\ t_1')\ (height\ t_2'), 0)\} \\
\Longrightarrow_{(\text{Abstraction})} & \{(t_1, node\ t_1'\ t_2'), (max\ x\ y, 0), (height\ t_1', x), (height\ t_2', y)\} \\
 & \ldots
\end{array}
$$

Partial sharing analysis can sometimes be expensive, if the number of solvable matching pairs is quite large and there are rarely similar pairs. Take the matching problem $\{(max\ x\ y,\ succ(succ(\cdots(succ\ 0)\cdots)))\}$ as an example, the solution list will record the matches of $(max\ x\ y,\ succ(succ(\cdots(succ\ 0)\cdots)))$, $\cdots$, $(max\ x\ y,\ succ\ 0)$, and $(max\ x\ y,\ 0)$, but no similar pairs. Nevertheless, according to our experiments, partial sharing analysis is useful in many cases. Especially, it is of benefit for functions which are defined through functions whose results are only dependent on some of their arguments like 'max', or for functions whose results are not affected by the order of arguments like '+', '*' and 'max'.

2.3.3 Commutative Matching

Some matching combinators may be commutative, as for example $+$, $*$, max:

$$
\begin{array}{lll}
x + y & =_{\mathcal{E}_+} & y + x \\
x * y & =_{\mathcal{E}_*} & y * x \\
max(x, y) & =_{\mathcal{E}_{max}} & max(y, x)
\end{array}
$$

Rewriting modulo a congruence is discussed by Peterson and Stickel in [65], and by Jouannaud and Kirchner in [43]. We are especially interested in rewriting modulo commutativity. We say that a term s **commutatively matches** (short: C-**matches**) a term t if there is a substitution σ such that $\sigma s =_C t$. For instance, $(succ\ x) + 0$ C-matches $0 + y$, since we have $(succ\ x) + 0 =_C 0 + (succ\ x)$.

Let $\mathcal{E}_f$ be a set of equations defining commutative function f. $(\mathcal{E}_f)_C$ denotes the set of all equations $s = t$, such that $s =_C \sigma l$ and $t = \sigma r$, for some rule $l = r$ in $\mathcal{E}_f$ and some substitution σ. The congruence relation induced by $(\mathcal{E}_f)_C$ generalizes $=_{\mathcal{E}_f}$. A term t is reducible by $(\mathcal{E}_f)_C$ iff some subterm of t C-matches a left-hand side of an equation in $\mathcal{E}$; otherwise it is called a $(\mathcal{E}_f)_C$-**normal form**.

By considering rewriting modulo commutativity, some semantic matching problems can be reduced to their solutions. For instance, if we know that '+' is commutative, then a matching pair like $(x + 0, t)$ can be reduced to (x, t), since

$$x + 0 \ =_C \ 0 + x \ =_{\mathcal{E}} \ x;$$

or a matching pair like $(x + succ\, y, succ(succ\, 0))$ can be transformed to the "simpler" one $(x + y, succ\, 0)$. That is to say, using rewriting modulo commutativity, we can transform some narrowing steps into rewriting steps. The former are more expensive than the latter.

On the other hand, for a commutative function, its two arguments are symmetric. Let $(f\ x\ y, t)$ be a matching pair with f a commutative function, and σ a match, then $\sigma' = \{x \mapsto \sigma y,\, y \mapsto \sigma x\}$ is also a match of the pair. So for such matching pairs we only need to find half of their matches and the rest can be obtained symmetrically.

The central idea to realize **commutative matching** is to transform the set of equation $\mathcal{E}_f$ of a commutative function f to its commutative counterpart $\mathcal{E}_f^C$, which satisfies the following conditions:

Correctness. $s =_{\mathcal{E}} t$, where s and t are ground terms, iff $s =_{(\mathcal{E}_f^C)_C} t$, where $(\mathcal{E}_f^C)_C$ is $\mathcal{E}_f^C$ modulo commutativity.

Redundance-freedom. If σ is a match of $\{(f\ x\ y, t)\}$ produced by a matching algorithm with respect to $\mathcal{E}_f^C$, $\sigma x \neq \sigma y$, then $\{x \mapsto \sigma y,\, y \mapsto \sigma x\}$ will not be produced.

For example, $\mathcal{E}_+^C$ and $\mathcal{E}_*^C$ can be presented as:

$$\begin{array}{lcl} 0 + y & = & y \\ (succ\ x) + (succ\ y) & = & succ(succ(x + y)) \\ \\ 0 * y & = & 0 \\ (succ\ x) * (succ\ y) & = & succ(y + (x + (x * y))). \end{array}$$

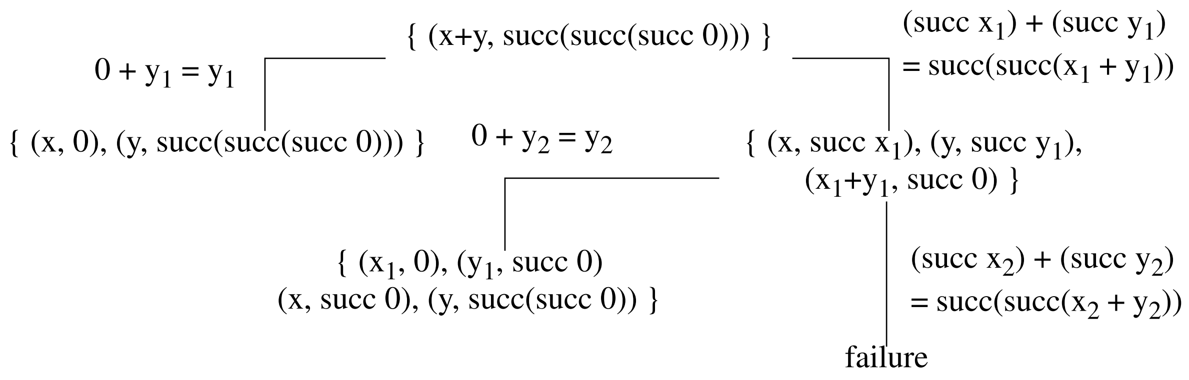

Figure 6: A matching tree produced by $\mathcal{MA}^1_c$

Figure 6 shows a matching tree of the matching problem $\{(x+y,\ succ(succ(succ\ 0)))\}$ constructed by using $\mathcal{E}_+^C$. It produces the solutions $\{x \mapsto 0,\ y \mapsto succ(succ(succ\ 0))\}$ and $\{x \mapsto succ\ 0,\ y \mapsto succ(succ\ 0)\}$; the solutions $\{x \mapsto succ(succ(succ\ 0)),\ y \mapsto 0\}$ and $\{x \mapsto succ(succ\ 0),\ y \mapsto succ\ 0\}$ can be obtained symmetrically.

The correctness of $\mathcal{E}_+^C$ is not difficult to show. Without loss of generality, let $s = a_1 + a_2$, where a_1, a_2 are ground terms.

- If $a_1 = 0$, then $0 + a_2 =_{\mathcal{E}_+} a_2$ and $0 + a_2 =_{\mathcal{E}_+^C} a_2$.
- If $a_2 = 0$, then $a_1 + 0 =_{\mathcal{E}_+} a_1$ and $a_1 + 0 =_C 0 + a_1 =_{\mathcal{E}_+^C} = a_1$.
- If $a_1 = succ\ a_1'$ and $a_2 = succ\ a_2'$, then it can be proved by using the induction hypothesis. □

In figure 7, we prove that $\mathcal{E}_+^C$ is redundance-free with respect to $\mathcal{MA}^1$ by structural induction on a, where $\{(x+y, a)\}$ is a matching problem, $a \in \mathcal{T}_{nat}$. For $\mathcal{E}_*^C$, the two conditions can be proved similarly.

Commutative matching is only useful for the functions that are used very often, because to use it, one should first transform a set of equations into its commutative counterpart. In $\mathcal{MA}_c^1$, we have implemented commutative matching for a set of predefined functions like '$+$', '$*$' and 'max', and rewriting modulo commutativity as well.

Case $a = 0$:

$\{ (x+y, 0) \}$

$0 + y_1 = y_1$

$\{ (x, 0), (y, 0), (y_1, 0) \}$

$(succ\ x_1) + (succ\ y_1) = succ(succ(x_1 + y_1))$

$\{ (x, succ\ x_1), (y, succ\ y_1), (succ(succ(x_1 + y_1)), 0)\}$

failure

Case $a = succ\ 0$:

$\{ (x+y, succ\ 0) \}$

$0 + y_1 = y_1$

$\{ (x, 0), (y, succ\ 0), (y_1, succ\ 0) \}$

$(succ\ x_1) + (succ\ y_1) = succ(succ(x_1 + y_1))$

$\{ (x, succ\ x_1), (y, succ\ y_1), (succ(succ(x_1 + y_1)), succ\ 0)\}$

failure

Case $a = succ(succ\ a_1)$:

$\{ (x+y, succ(succ\ a_1)) \}$

$0 + y_1 = y_1$

$\{ (x, 0), (y, succ(succ\ a_1)), (y_1, succ(succ\ a_1)) \}$

$(succ\ x_1) + (succ\ y_1) = succ(succ(x_1 + y_1))$

$\{ (x, succ\ x_1), (y, succ\ y_1), (succ(succ(x_1 + y_1)), succ(succ\ a_1))\}$

SIMPL

$\{ (x, succ\ x_1), (y, succ\ y_1), (x_1 + y_1, a_1)\}$

Figure 7: Proof of the redundance-freedom of $\mathcal{E}^C_+$

2.3.4 Analysis and Test Results

Algorithm analysis

The improvement techniques also offer a direct way to analyse the algorithm, particularly dependency analysis, because we can treat each solvable matching pair independently. For simplicity, we ignore the simplification steps.

To solve the matching pair $(f\ t_1 \cdots t_n, t)$ with respect to a constructor-based specification $\langle \Sigma, \mathcal{E} \rangle$, where f is a matching combinator in Σ, the following items could be of interest for efficiency considerations:

- The number M_f of equations in $\mathcal{E}_f$. There are at most M_f equations that can be used to narrow the matching pair, i.e., the number of new matching problems produced by applying all possible (Narrowing) is at most M_f. For instance, M_{max} = 3, $M_{height} = 2$. Take $max\ x\ y$ as an example, which can be structurally narrowed to three new terms.

$$\begin{array}{lll} max\ x\ y & \rightsquigarrow_{max\ 0\ y=y} & y_1 \\ max\ x\ y & \rightsquigarrow_{max\ (succ\ x)\ 0=succ\ x} & succ\ x_1 \\ max\ x\ y & \rightsquigarrow_{max\ (succ\ x)\ (succ\ y)=succ(max\ x\ y)} & succ(max\ x_1\ y_1) \end{array}$$

- The multiset S_r of function symbols that occur on the right-hand side of the equation $l = r$. It determines the number of matching pairs in a new problem after applying (Narrowing) with $l = r$ and possible simplification steps. For instance,

$$\begin{array}{lll} S_y & = & \emptyset, \\ S_{succ\ x} & = & \emptyset, \\ S_{succ(max\ x\ y)} & = & \{max\} \\ S_0 & = & \emptyset, \\ S_{succ(max\ (height\ x)\ (height\ y))} & = & \{max, height, height\}. \end{array}$$

Take $\{(height\ t, succ(succ\ 0))\}$ as an example,

$$\begin{array}{ll} & \{(height\ t, succ\ 0)\} \\ \Longrightarrow_{\text{(Narrowing)}} & \{(t, node\ t_1\ t_2), (succ(max\ (height\ t_1)\ (height\ t_2)), \\ & succ(succ\ 0))\} \\ \Longrightarrow_{\text{(Decomposition)}} & \{(t, node\ t_1\ t_2), (max\ (height\ t_1)\ (height\ t_2), 0)\} \\ \Longrightarrow_{\text{(Abstraction)}} & \{(t, node\ t_1\ t_2), (max\ x\ y, 0), (height\ t_1, x), (height\ t_2, y)\} \end{array}$$

Now the matching problem contains three matching pairs to be solved.

- The depth of t. It determines how many times the rule (Narrowing) can be applied, because after some (Narrowing) transformations, some (Decomposition) transformations will follow, which reduce the depth of t. On the other hand, the application of (Abstraction) introduces dependent pairs. If (t_1, s_1) is dependent on (t_2, s_2), then $depth(s_1)$ depends on the matches of (t_2, s_2), such as $depth(x)$ and $depth(y)$ in $(height\ t_1, x)$ and $(height\ t_2, y)$ depend on the matches of $(max\ x\ y, 0)$, see the above example. Because max is a proper non-decreasing function, so $depth(x)$, $depth(y)$ $\leq depth(0)$.

We use $\mathcal{T}_f(m)$ to denote the time upper-bound to solve the matching pair ($f\ t_1 \cdots\ t_n$, t), where $depth(t) = m$ and each t_i ($1 \leq i \leq n$) contains no function symbols. $\mathcal{T}_f(m)$ exactly gives the upper-bound of the number of narrowing steps to solve the matching pair. The number of matches of the matching pair will also never exceed $\mathcal{T}_f(m)$. Let C be the constant cost of a narrowing step.

Generally, $\mathcal{T}_f(m)$ is exponential. But if f satisfies the following conditions, $\mathcal{T}_f(m)$ is linear, in this case f is called a **linear function**.

- If $f\ l_1 \cdots l_n = r$ is an equation, then r contains only one occurrence of a function symbol; and
- the function symbol in r is f, or it is another linear function.

$+$, max and $++$ are linear functions. The following equations define $T_+(m)$ recursively:

$$\begin{array}{lcll} \mathcal{T}_+(0) & = & 2*C & (1) \\ \mathcal{T}_+(m+1) & = & 2*C+\mathcal{T}_+(m) & (2) \end{array}$$

If $m = 0$, we only need two narrowing steps (1), since there are two equations in $\mathcal{E}_+$; if $m > 0$ then, in addition to two narrowing steps, we should solve one new matching problem with one matching pair (2), since $S_{succ(x+y)} = \{+\}$. So $\mathcal{T}_+(m) = 2*C*m$, i.e., $\mathcal{O}(m)$.

f is called a **polynomial function** if f is linear, or f satisfies the following conditions:

- If $f\ l_1 \cdots l_n = r$, then f does not occur in r; and
- if $g_1, \cdots, g_m$ are the function symbols in r, then they are all polynomial functions.

Polynomial functions, which are not linear, are functions not defined recursively by themselves. The following function $addp : nat \rightarrow pair \rightarrow pair$ is a polynomial function.

$$addp\ n\ (x,y) \quad = \quad (n+x, n+y)$$

$$\mathcal{T}_{addp}(m) \quad = \quad C + \mathcal{T}_+(m-1) * \mathcal{T}_+(m-1) \quad (3)$$

To solve the matching pair ($addp\ n\ (x,y), (t,s)$), in addition to a narrowing step, we will solve two new matching problems associated with $+$ (3). So $\mathcal{T}_{addp}(m)$ is $\mathcal{O}(m^2)$.

All acyclic matching combinators are called **exponential functions**. For instance, $*$ and $height$ are exponential functions.

$$\begin{array}{lcll} \mathcal{T}_*(0) & = & 3*C & (4) \\ \mathcal{T}_*(m+1) & = & 3*C+\mathcal{T}_+(m)*(1+\mathcal{T}_*(m)) & (5) \end{array}$$

If $m = 0$, only three narrowing steps are necessary (4), since there are three equations in $\mathcal{E}_*$; otherwise we should solve a new matching problem with two matching pairs, the occurrence of $*$ is nested below $+$ (5), since $S_{succ(y+(x*(succ\ y)))} = \{+, *\}$. So $\mathcal{T}_*$ is obviously exponential.

Test results

To summarize, dependency analysis is the most important technique to make the matching algorithm efficient. Matching trees constructed by $\mathcal{M}A^1_d$ become much smaller than those constructed by $\mathcal{M}A^1$. Useless narrowing steps are avoided, and each variable in a matching problem is instantiated only once. Different from the dependency analysis, partial sharing analysis and the use of the commutative property are helpful for some special matching problems. Especially, partial sharing analysis can sometimes be expensive. We doubt the existence of a matching algorithm with polynomial complexity for general matching combinators. Remember that there are 651 trees of height 4, the original matching pair is just $\{(height(node\ x\ y), succ(succ(succ(succ\ 0))))\}$.

The following table gives the test results of the time-costs (in "minutes : seconds") to solve the matching problems (1) $\{(height(node(t_1, t_2)), 4)\}$ and (2) $\{(x * y, 36)\}$ by using $\mathcal{M}A^1$, $\mathcal{M}A^1_d$, $\mathcal{M}A^1_p$ and $\mathcal{M}A^1_c$ respectively, where 4 and 36 are abbreviations of natural numbers presented by *succ* and 0. These examples are chosen in order to show the usefulness of all these techniques.

	$\mathcal{M}A^1$	$\mathcal{M}A^1_d$	$\mathcal{M}A^1_p$	$\mathcal{M}A^1_c$
(1)	57 : 22	4 : 26	3 : 0	2 : 40
(2)	4 : 20	1 : 45	1 : 30	0 : 55

2.4 Correctness of the Matching Algorithm

The correctness of the matching algorithm will be shown in two pieces: a proof of partial correctness and a proof of termination. We will concentrate on the correctness of $\mathcal{TS}_d^1$, since it is the base of the matching algorithms $\mathcal{MA}_d^1$, $\mathcal{MA}_p^1$ and $\mathcal{MA}_c^1$. We always present the global results at first, which will be proved through a sequences of lemmas and theorems.

2.4.1 Soundness and Completeness

Theorem 2.2 The matching algorithm is sound and complete, i.e., $\sigma \in \mathcal{M}(\Gamma)$, iff there exists a sequence of transformations $\Gamma \Longrightarrow^*_{\mathcal{TS}_d^1} \Gamma'$ such that $\sigma_{\Gamma'} \leq_{[\mathcal{V}(\Gamma)]} \sigma$, where $\sigma_{\Gamma'}$ is the substitution formed from Γ'. □

The central point of the correctness proof lies in the fact that the algorithm always chooses a solvable pair to obtain some intermediate solutions, through the applications of the rules (Variable) and (Narrowing) some new solvable pairs may be produced, trying then to obtain some new solutions, until all pairs are in solved form.

It has already been mentioned that a ground matching problem could be transformed to a non-ground one by using (Abstraction) and (Narrowing), so we should extend the definition of sets of matchs ($\mathcal{M}$) to cover the case of non-ground matching problems.

Let $\Gamma \overset{rep}{=} \Gamma_0 \,\&\, \Gamma_1 \,\&\, \Gamma_2$, $\sigma \in \mathcal{M}'(\Gamma)$ is defined as follows: there exist $\sigma_1 \in \mathcal{M}(\Gamma_1)$ and $\sigma_2 \in \mathcal{M}(\sigma_1\Gamma_2)$ such that $\sigma = \sigma_2 \circ \sigma_1 \circ \sigma_{\Gamma_0}$.

We have said that the algorithm begins with a ground matching problem Γ and terminates with a matching problem Γ' in solved form. Hence, $\Gamma \overset{rep}{=} \{\} \,\&\, \Gamma \,\&\, \{\}$, and $\Gamma' \overset{rep}{=} \Gamma' \,\&\, \{\} \,\&\, \{\}$. $\mathcal{M}'(\Gamma) = \mathcal{M}(\Gamma)$ and $\mathcal{M}'(\Gamma') = \mathcal{M}(\Gamma')$.

Definition 2.20 Let the following be a transformation of $\mathcal{TS}_d^1$ issued by rule r

$$\Gamma \overset{rep}{=} \Gamma_0 \,\&\, \Gamma_1 \,\&\, \Gamma_2 \Longrightarrow_r \Gamma_0' \,\&\, \Gamma_1' \,\&\, \Gamma_2' \overset{rep}{=} \Gamma'.$$

- It is **sound**, if for all $\theta \in \mathcal{M}'(\Gamma')$, there exists $\sigma \in \mathcal{M}'(\Gamma)$ such that $\sigma =_{[\mathcal{V}(\Gamma)]} \theta$.
- It is **complete**, if for all $\sigma \in \mathcal{M}'(\Gamma)$, there exists $\theta \in \mathcal{M}'(\Gamma')$ such that $\theta \leq_{[\mathcal{V}(\Gamma)]} \sigma$.
- It is **correct**, if it is both sound and complete.
- Rule r is **correct**, if every transformation issued by the rule is correct.

It is easy to see that (Remove), (Deletion) and (Decomposition) are correct rules in $\mathcal{TS}_d^1$, so we only need to prove that (Variable), (Abstraction) and (Narrowing) are also correct. Theorem 2.1 ensures that if $\Gamma \overset{rep}{=} \Gamma_0\&\Gamma_1\&\Gamma_2$ is not in solved form, then $\Gamma_1 \neq \emptyset$. In the proofs we will often use the fact that only idempotent substitutions are considered.

Lemma 2.7 (Variable) (Variable) is a correct rule in $\mathcal{TS}_d^1$.

Proof. Consider the transformation:

$$\Gamma \overset{rep}{=} \Gamma_0 \,\&\, \{(x,t)\}\cup\Gamma_1 \,\&\, \Gamma_2 \Longrightarrow_{\text{(Variable)}} \{(x,t)\}\cup(\sigma\Gamma_0) \,\&\, (\sigma\Gamma_1)\cup(\sigma\Gamma_{2,1}) \,\&\, (\sigma\Gamma_{2,2}) \overset{rep}{=} \Gamma',$$

where $\sigma = \{x \mapsto t\}$ and $\Gamma_{2,1} \cup \Gamma_{2,2} = \Gamma_2$. We show that $\mathcal{M}'(\Gamma) = \mathcal{M}'(\Gamma')$.

- Soundness.

  Let $\theta_2 \circ \theta_1 \circ (\sigma \circ \sigma_{\Gamma_0}) \in \mathcal{M}'(\Gamma')$ such that $\theta_1 \in \mathcal{M}((\sigma\Gamma_1) \cup (\sigma\Gamma_{2,1}))$ and $\theta_2 \in \mathcal{M}(\theta_1(\sigma\Gamma_{2,2}))$. Note that $\{(x,t)\} \cup (\sigma\Gamma_0)$ forms the substitution $\sigma \circ \sigma_{\Gamma_0}$.

  We split θ_1 into two substitutions: $\tau_1 = \theta_1[\mathcal{V}(\sigma\Gamma_1)]$ and $\tau_2 = \theta_1[\mathcal{V}(\sigma\Gamma_{2,1})]$. Because θ_1 is idempotent, $\tau_2 \circ \tau_1 = \theta_1$. Define $\sigma_1 = \tau_1 \circ \sigma$ and $\sigma_2 = \theta_2 \circ \tau_2$. It is obvious that $\sigma_2 \circ \sigma_1 \circ \sigma_{\Gamma_0} = \theta_2 \circ \theta_1 \circ \sigma \circ \sigma_{\Gamma_0}$. We should prove that $\sigma_2 \circ \sigma_1 \circ \sigma_{\Gamma_0} \in \mathcal{M}'(\Gamma)$, i.e., $\sigma_1 \in \mathcal{M}(\{(x,t)\} \cup \Gamma_1)$ and $\sigma_2 \in \mathcal{M}(\sigma_1\Gamma_2)$.

  We show first that $\sigma_1 \in \mathcal{M}(\{(x,t)\} \cup \Gamma_1)$.

  $$\begin{array}{rll} & \sigma_1 x = (\tau_1 \circ \sigma)x & (\text{definition of } \sigma_1) \\ = & \tau_1 t & (\sigma = \{x \mapsto t\}) \\ = & t & ((x,t) \text{ is solvable}) \end{array}$$

  Let $(s,s') \in \Gamma_1$,

  $$\begin{array}{rll} & \sigma_1 s = (\tau_1 \circ \sigma)s & (\text{definition of } \sigma_1) \\ = & \sigma s' & (\tau_1(\sigma s) = \sigma s') \\ = & s' & ((s,s') \text{ is solvable}, x \notin \mathcal{V}(s')) \end{array}$$

  Now we show $\sigma_2 \in \mathcal{M}(\sigma_1\Gamma_2)$. Let $(s,s') \in \Gamma_{2,1}$,

  $$\begin{array}{rll} & \sigma_2(\sigma_1 s) = (\theta_2 \circ \theta_1)(\sigma s) & (\text{definition of } \sigma_1, \sigma_2) \\ = & \theta_2(\sigma s') & (\theta_1 \in \mathcal{M}((\sigma\Gamma_1) \cup (\sigma\Gamma_{2,1}))) \\ = & \sigma s' & ((\sigma s, \sigma s') \text{ is solvable in } \Gamma') \\ = & \theta_1(\sigma s') & ((\sigma s, \sigma s') \text{ is solvable in } \Gamma') \\ = & \sigma_1 s' & (\text{definition of } \sigma_1) \end{array}$$

  Let $(s,s') \in \Gamma_{2,2}$, then

  $$\begin{array}{rll} & \sigma_2(\sigma_1 s) = (\theta_2 \circ \theta_1)(\sigma s) & (\text{definition of } \sigma_1, \sigma_2) \\ = & \theta_1(\sigma s') & (\theta_2 \in \mathcal{M}(\theta_1(\sigma\Gamma_{2,2}))) \\ = & \sigma_1 s' & (\text{definition of } \sigma_1) \end{array}$$

- Completeness.

  Let $\sigma_2 \circ \sigma_1 \circ \sigma_{\Gamma_0} \in \mathcal{M}'(\Gamma)$ such that $\sigma_1 \in \mathcal{M}(\{(x,t)\} \cup \Gamma_1)$ and $\sigma_2 \in \mathcal{M}(\sigma_1\Gamma_2)$.

  We split σ_2 into two substitutions: $\tau_1 = \sigma_2[\mathcal{V}(\sigma_1\Gamma_{2,1})]$, $\theta_2 = \sigma_2[\mathcal{V}(\sigma_1\Gamma_{2,2})]$. Because σ_2 is idempotent, $\tau_1 \circ \theta_2 = \sigma_2$. Define $\theta_1 = \tau_1 \circ \sigma_1$. Obviously, $\theta_2 \circ \theta_1 \circ \sigma \circ \sigma_{\Gamma_0} = \sigma_2 \circ \sigma_1 \circ \sigma_{\Gamma_0}$, note that $\sigma_1 \circ \sigma = \sigma_1$. We should prove $\theta_2 \circ \theta_1 \circ \sigma \circ \sigma_{\Gamma_0} \in \mathcal{M}'(\Gamma')$, i.e., $\theta_1 \in \mathcal{M}((\sigma\Gamma_1) \cup (\sigma\Gamma_{2,1}))$ and $\theta_2 \in \mathcal{M}(\theta_1(\sigma\Gamma_{2,2}))$.

  We show first $\theta_2 \in \mathcal{M}(\theta_1(\sigma\Gamma_{2,2}))$. Let $(s,s') \in \Gamma_{2,2}$,

  $$\begin{array}{rll} & \theta_2(\theta_1(\sigma s) = (\sigma_2 \circ \sigma_1)(\sigma s) & (\text{definition of } \theta_1, \theta_2) \\ = & (\sigma_2 \circ \sigma_1)s & (\sigma_1 \circ \sigma = \sigma_1) \\ = & \sigma_1 s' & (\sigma_2 \in \mathcal{M}(\sigma_1\Gamma_2)) \\ = & (\tau_1 \circ \sigma_1)s' & (\mathcal{V}(\sigma_1\Gamma_{2,1}) \cap \mathcal{V}(\sigma_1 s') = \emptyset) \\ = & \theta_1(\sigma s') & (\text{definition of } \theta_1) \end{array}$$

Now we show $\theta_1 \in \mathcal{M}((\sigma\Gamma_1) \cup (\sigma\Gamma_{2,1}))$. Let $(s, s') \in \Gamma_{2,1}$,

$$\begin{array}{lll} & \theta_1(\sigma s) = (\tau_1 \circ \sigma_1)s & (\text{definition of } \theta_1) \\ = & \sigma_1 s' & (\sigma_2 \in \mathcal{M}(\sigma_1\Gamma_2), \text{definition of } \tau_1) \\ = & \sigma s' & (\mathcal{V}(\sigma s') \cap \mathcal{V}(\Gamma_1) = \emptyset) \end{array}$$

If $(s, s') \in \Gamma_1$, then

$$\begin{array}{lll} & \theta_1(\sigma s) = (\tau_1 \circ \sigma_1)s & (\text{definition of } \theta_1) \\ = & \tau_1 s' & (\sigma_1 \in \mathcal{M}(\Gamma_1)) \\ = & s' & (\mathcal{D}(\tau_1) \cap \mathcal{V}(s') = \emptyset) \\ = & \sigma s' & (x \notin \mathcal{V}(s')) \end{array}$$

□

Lemma 2.8 (Abstraction) (Abstraction) is a correct rule in $\mathcal{TS}^1_d$.

Proof. Consider the transformation:

$$\Gamma \stackrel{rep}{=} \Gamma_0 \ \& \ \{(t|_{\{p_1,\cdots,p_k\}}, s)\} \cup \Gamma_1 \ \& \ \Gamma_2 \Longrightarrow_{(\text{Abstraction})}$$
$$\Gamma_0 \ \& \ \{(t[x_1,\cdots,x_k]_{\{p_1,\cdots,p_k\}}, s)\} \cup \Gamma_1 \ \& \ \{(t|_{p_1}, x_1), \cdots, (t|_{p_k}, x_k)\} \cup \Gamma_2 \stackrel{rep}{=} \Gamma',$$

where all x_i's are new variables. We prove that $\mathcal{M}'(\Gamma) =_{[\mathcal{V}(\Gamma)]} \mathcal{M}'(\Gamma')$.

- Soundness.

  Let $\theta_2 \circ \theta_1 \circ \sigma_{\Gamma_0} \in \mathcal{M}'(\Gamma')$ such that

  - $\theta_1 \in \mathcal{M}(\{(t[x_1,\cdots,x_k]_{\{p_1,\cdots,p_k\}}, s)\} \cup \Gamma_1)$, and
  - $\theta_2 \in \mathcal{M}(\theta_1(\{(t|_{p_1}, x_1), \cdots, (t|_{p_k}, x_k)\} \cup \Gamma_2))$.

  Let $W = \mathcal{V}(\theta_1(t|_{p_1})) \cup \cdots \cup \mathcal{V}(\theta_1(t|_{p_k})) \cup \mathcal{V}(\theta_1 x_1) \cup \cdots \cup \mathcal{V}(\theta_1 x_k)$. We first split θ_2 into two substitution: $\tau_1 = \theta_2\ [W]$, and $\sigma_2 = \theta_2\ [\mathcal{V}(\theta_1\Gamma_2)]$. Because θ_2 is idempotent, $\sigma_2 \circ \tau_1 = \theta_2$. Define $\sigma_1 = \tau_1 \circ \theta_1$. Obviously, $\sigma_2 \circ \sigma_1 \circ \sigma_{\Gamma_0} = \theta_2 \circ \theta_1 \circ \sigma_{\Gamma_0}$.

  It is not difficult to see that $\sigma_2 \in \mathcal{M}(\sigma_1\Gamma_2)$. Now we show $\sigma_1 \in \mathcal{M}(\{(t|_{\{p_1,\cdots,p_k\}}, s)\} \cup \Gamma_1)$

$$\begin{array}{lll} & \sigma_1(t|_{\{p_1,\cdots,p_k\}}) & \\ = & (\tau_1 \circ \theta_1)(t|_{\{p_1,\cdots,p_k\}}) & (\text{definition of } \sigma_1) \\ = & \tau_1(\theta_1 t)[\tau_1(\theta_1(t|_{p_1})), \cdots, \tau_1(\theta_1(t|_{p_k}))]_{\{p_1,\cdots,p_k\}} & \\ = & \tau_1(\theta_1 t)[\theta_1 x_1, \cdots, \theta_1 x_k]_{\{p_1,\cdots,p_k\}} & (\theta_2 \in \mathcal{M}(\theta_1(\{(t|_{p_1}, x_1), \cdots, \\ & & (t|_{p_k}, x_k)\} \cup \Gamma_2))) \\ = & \tau_1(\theta_1(t[x_1,\cdots,x_k]_{\{p_1,\cdots,p_k\}})) & \\ = & \tau_1 s & (\theta_1 \in \mathcal{M}(\{(t[x_1,\cdots,x_k]_{\{p_1,\cdots,p_k\}}, s)\} \\ & & \cup\Gamma_1)) \\ = & s & (\mathcal{V}(s) \cap W = \emptyset) \end{array}$$

  If $(s, s') \in \Gamma_1$, then

$$\begin{array}{lll} & \sigma_1 s = \tau_1(\theta_1 s) & \\ =_{\mathcal{E}} & \tau_1 s' & (\theta_1 \in \mathcal{M}(\{(t[x_1,\cdots,x_k]_{\{p_1,\cdots,p_k\}}, s)\} \cup \Gamma_1)) \\ = & s' & ((s, s') \text{ is solvable in } \Gamma) \end{array}$$

- Completeness.

  Let $\sigma_2 \circ \sigma_1 \circ \sigma_{\Gamma_0} \in \mathcal{M}'(\Gamma)$ such that $\sigma_1 \in \mathcal{M}(\{(t|_{\{p_1,\cdots,p_k\}}, s)\} \cup \Gamma_1)$ and $\sigma_2 \in \mathcal{M}(\sigma_1\Gamma_2)$.

  Define $\theta_1 = \sigma_1 \circ \{x_1 \mapsto t|_{p_1}, \cdots, x_k \mapsto t|_{p_k}\}$ and $\theta_2 = \sigma_2$. Obviously, $\theta_2 \circ \theta_1 \circ \sigma_{\Gamma_0} =_{[\mathcal{V}(\Gamma)]} \sigma_2 \circ \sigma_1 \circ \sigma_{\Gamma_0}\ [\mathcal{V}(\Gamma)]$.

  It is not difficult to see that $\theta_2 \in \mathcal{M}(\theta_1\Gamma_{2,2})$. Now we show $\theta_1 \in \mathcal{M}(\{(t[x_1, \cdots, x_k]_{\{p_1,\cdots,p_k\}}, s)\} \cup \Gamma_1)$.

$$\begin{array}{ll} \theta_1(t[x_1, \cdots, x_k]_{\{p_1,\cdots,p_k\}}) & \\ = \sigma_1(t[t|_{p_1}, \cdots, t|_{p_k}]_{\{p_1,\cdots,p_k\}}) & \text{(definition of } \theta_1) \\ = \sigma_1(t|_{\{p_1,\cdots,p_k\}}) & \\ = s & (\sigma_1 \in \mathcal{M}(\{(t|_{\{p_1,\cdots,p_k\}}, s)\} \cup \Gamma_1)) \end{array}$$

  If $(s, s') \in \Gamma_1$, then

$$\begin{array}{lll} & \theta_1 s = \sigma_1 s & \\ =_{\mathcal{E}} & s' & (\theta_1 \in \mathcal{M}(\{(t[x_1, \cdots, x_k]_{\{p_1,\cdots,p_k\}}, s)\} \cup \Gamma_1)) \end{array}$$

□

Lemma 2.9 (Narrowing) (Narrowing) is a correct rule in $\mathcal{TS}_d^1$.

Proof. Consider the transformation:

$$\begin{array}{l} \Gamma \stackrel{rep}{=} \Gamma_0 \ \&\ \{(f\ t_1 \cdots\ t_n, t)\} \cup \Gamma_1 \ \&\ \Gamma_2 \\ \Longrightarrow_{(\text{Narrowing})} \Gamma_\sigma \cup (\sigma\Gamma_0) \ \&\ \{(\sigma r, t)\} \cup (\sigma\Gamma_1) \cup (\sigma\Gamma_{2,1}) \ \&\ (\sigma\Gamma_{2,2}) \stackrel{rep}{=} \Gamma', \end{array}$$

where $\Gamma_{2,1} \cup \Gamma_{2,2} = \Gamma_2$, $l = r \in \mathcal{E}$, $\sigma = mgu(f\ t_1 \cdots\ t_n, l)$.

- Soundness: For every $\theta \in \mathcal{M}'(\Gamma')$, there exists a $\sigma \in \mathcal{M}'(\Gamma)$ such that $\theta =_{[\mathcal{V}(\Gamma)]} \sigma$.

  Let $\theta_2 \circ \theta_1 \circ (\sigma \circ \sigma_{\Gamma_0}) \in \mathcal{M}'(\Gamma')$ such that

  - $\theta_1 \in \mathcal{M}(\{(\sigma r, t)\} \cup (\sigma\Gamma_1) \cup (\sigma\Gamma_{2,1}))$, and
  - $\theta_2 \in \mathcal{M}(\theta_1(\sigma\Gamma_{2,2}))$.

  We split θ_1 into two substitutions: $\tau_1 = \theta_1[\mathcal{V}(\sigma r, t) \cup \mathcal{V}(\sigma\Gamma_1)]$ and $\tau_2 = \theta_1[\mathcal{V}(\sigma\Gamma_{2,1})]$. Because θ_1 is idempotent, $\tau_2 \circ \tau_1 = \theta_1$. Define $\sigma_1 = \tau_1 \circ \sigma$, $\sigma_2 = \theta_2 \circ \tau_2$. Obviously, $\sigma_2 \circ \sigma_1 \circ \sigma_{\Gamma_0} = \theta_2 \circ \theta_1 \circ (\sigma \circ \sigma_{\Gamma_0})$.

  We show first that $\sigma_2 \in \mathcal{M}(\sigma_1\Gamma_2)$. Let $(s, s') \in \Gamma_{2,1}$,

$$\begin{array}{lll} & \sigma_2(\sigma_1 s) = \theta_2(\theta_1(\sigma s)) & \text{(definition of } \sigma_1, \sigma_2) \\ = & \theta_2(\sigma s') & \text{(definition of } \theta_1) \\ = & \sigma s' & ((\sigma s, \sigma s') \text{ is solvable in } \Gamma') \\ = & (\tau_1 \circ \sigma) s' & ((\sigma s, \sigma s') \text{ is solvable in } \Gamma') \\ = & \sigma_1 s' & \text{(definition of } \sigma_1) \end{array}$$

Let $(s, s') \in \Gamma_{2,2}$,

$$\begin{array}{lll} & \sigma_2(\sigma_1 s) = \theta_2(\theta_1(\sigma s)) & (\text{definition of } \sigma_1, \sigma_2) \\ = & \theta_1(\sigma s') & (\theta_2 \in \mathcal{M}(\theta_1(\sigma\Gamma_{2,2}))) \\ = & (\tau_2 \circ \tau_1)(\sigma s') & (\text{definition of } \tau_1\ \tau_2) \\ = & \tau_1(\sigma s') & (\mathcal{D}(\tau_2) \cap \mathcal{V}(\sigma s') = \emptyset) \\ = & \sigma_1 s' & (\text{definition of } \sigma_1) \end{array}$$

Now we show $\sigma_1 \in \mathcal{M}(\{(f\ t_1 \cdots t_n, t)\} \cup \Gamma_1)$.

$$\begin{array}{lll} & \sigma_1(f\ t_1 \cdots t_n) & \\ = & (\tau_1 \circ \sigma)(f\ t_1 \cdots t_n) & \\ = & (\tau_1 \circ \sigma) l & (\sigma = mgu(f\ t_1 \cdots t_n, l)) \\ =_{\mathcal{E}} & (\tau_1 \circ \sigma) r & (l = r \in \mathcal{E}) \\ = & t & (\theta_1 \in \mathcal{M}(\{(\sigma r, t)\} \cup (\sigma\Gamma_1) \cup (\sigma\Gamma_{2,1}))) \end{array}$$

Let $(s, s') \in \Gamma_1$,

$$\begin{array}{lll} & \sigma_1 s = \tau_1(\sigma s) & \\ =_{\mathcal{E}} & \sigma s' & (\theta_1 \in \mathcal{M}(\{(\sigma r, t)\} \cup (\sigma\Gamma_1) \cup (\sigma\Gamma_{2,1}))) \\ = & s' & (\mathcal{V}(f\ t_1 \cdots t_n) \cap \mathcal{V}(s') = \emptyset) \end{array}$$

- Completeness: For every $\sigma \in \mathcal{M}'(\Gamma)$, there exists $\theta \in \mathcal{M}'(\Gamma')$, such that $\theta \leq_{[\mathcal{V}(\Gamma)]} \sigma$.

  Let $\sigma_2 \circ \sigma_1 \circ \sigma_{\Gamma_0} \in \mathcal{M}'(\Gamma)$ such that $\sigma_1 \in \mathcal{M}(\{(f\ t_1 \cdots t_n, t)\} \cup \Gamma_1)$ and $\sigma_2 \in \mathcal{M}(\sigma_1\Gamma_2)$.

  There exist $l = r \in \mathcal{E}$ and a substitution τ such that $f\ t_1 \cdots\ t_n$ and l are unifiable and $\sigma_1(f\ t_1 \cdots t_n) = \tau(f\ t_1 \cdots t_n) = \tau l =_{\mathcal{E}} \tau r =_{\mathcal{E}} t$. We split σ_2 into two substitutions: $\tau_1 = \sigma_2[\mathcal{V}(\sigma_1\Gamma_{2,1})]$ and $\theta_2 = \sigma_2[\mathcal{V}(\sigma_1\Gamma_{2,2})]$. Because σ_2 is idempotent, $\theta_2 \circ \tau_1 = \sigma_2$. Define $\tau_1' = \sigma_1[\mathcal{D}(\sigma_1) - \mathcal{D}(\sigma)]$, $\theta_1 = \tau_1 \circ \tau_1'$. Because $\sigma = mgu(f\ t_1 \cdots t_n, l)$, so $\tau_1' \circ \sigma \leq \sigma_1$. Consequently, $\theta_2 \circ \theta_1 \circ \sigma \circ \sigma_{\Gamma_0} \leq_{[\mathcal{V}(\Gamma)]} \sigma_2 \circ \sigma_1 \circ \sigma_{\Gamma_0}$.

  We show first $\theta_2 \in \mathcal{M}(\theta_1(\sigma\Gamma_{2,2}))$. Let $(s, s') \in \Gamma_{2,2}$,

$$\begin{array}{lll} & \theta_2(\theta_1(\sigma s)) & \\ = & \theta_2((\tau_1 \circ \tau_1')(\sigma s)) & (\text{definition of } \theta_1) \\ = & \sigma_2(\tau_1'(\sigma s)) & (\theta_2 \circ \tau_1 = \sigma_2) \\ = & \sigma_2(\sigma_1 s) & (\text{definition of } \tau_1') \\ = & \sigma_1 s' & (\text{definition of } \sigma_2) \\ = & \tau_1'(\sigma s') & (\text{definition of } \tau_1') \\ = & (\tau_1 \circ \tau_1')(\sigma s') & (\mathcal{D}(\tau_1) \cap \mathcal{V}(\sigma s') = \emptyset, \text{lemma 2.5}) \\ = & \theta_1(\sigma s') & (\text{definition of } \theta_1) \end{array}$$

  Now we show $\theta_1 \in \mathcal{M}(\{(\sigma r, t)\} \cup (\sigma\Gamma_1) \cup (\sigma\Gamma_{2,1}))$.

$$\begin{array}{lll} & \theta_1(\sigma r) & \\ =_{\mathcal{E}} & \theta_1(\sigma l) & (l = r \in \mathcal{E}) \\ = & \theta_1(\sigma(f\ t_1 \cdots t_n)) & (\sigma = mgu(f\ t_1 \cdots t_n, l)) \\ = & (\tau_1 \circ \tau_1')(\sigma(f\ t_1 \cdots t_n)) & (\text{definition of } \theta_1) \\ = & (\tau_1 \circ \sigma_1)(f\ t_1 \cdots t_n) & (\text{definition of } \tau_1') \\ = & \tau_1 t & (\sigma_1 \in \mathcal{M}(\{(f\ t_1 \cdots\ t_n, t)\} \cup \Gamma_1)) \\ = & t & (\mathcal{V}(t) \cap \mathcal{V}(\sigma\Gamma_{2,1}) = \emptyset) \end{array}$$

If $(s, s') \in \Gamma_1$, then

$$\begin{array}{lll} & \theta_1(\sigma s) & \\ = & (\tau_1 \circ \sigma_1)s & (\text{definition of } \theta_1) \\ = & \tau_1 s' & (\sigma_1 \in \mathcal{M}(\{(f\ t_1 \cdots t_n, t)\} \cup \Gamma_1)) \\ = & s' & (\mathcal{V}(s') \cap \mathcal{V}(\sigma\Gamma_{2,1}) = \emptyset) \end{array}$$

If $(s, s') \in \Gamma_{2,1}$, then

$$\begin{array}{lll} & \theta_1(\sigma s) & \\ = & (\tau_1 \circ \sigma_1)s & (\text{definition of } \theta_1) \\ = & \sigma_1 s' & (\sigma_2 \in \mathcal{M}(\sigma_1\Gamma_2), \text{definition of } \tau_1) \\ = & \sigma s' & (\mathcal{V}(\sigma s') \cap \mathcal{V}(\Gamma_1) = \emptyset) \end{array}$$

□

Via the induction over the length of transformation sequences and using the lemmas 2.7 – 2.9, theorem 2.2 can be proved. To prove the following theorem, we need the fact that a constructor-based specification is an orthogonal term rewriting system.

Theorem 2.3 (Minimality) Let Γ be a ground matching problem and $\{\}\ \&\ \Gamma\ \&\ \{\} \Longrightarrow^*_{\mathcal{TS}^1_d} \Gamma'\ \&\ \{\}\ \&\ \{\}$, then Γ' forms a minimal match of Γ.

Proof sketch. The only two rules to be considered are (Variable) and (Narrowing). Any application of the remaining rules does not influence the matches.

- Let (x, t) be a solvable matching pair in a matching problem, then the minimal match of the pair is $\{x \mapsto t\}$. That is, the (Variable) transformation produces a minimal partial solution.
- Let $(f\ t_1 \cdots t_n, t)$ be a solvable matching pair in a matching problem, $l_1 = r_1$ and $l_2 = r_2$ be two equations applied to narrow it. If $\sigma_1 = mgu(f\ t_1 \cdots t_n, l_1)$ and $\sigma_2 = mgu(f\ t_1 \cdots t_n, l_2)$, then $\sigma_1 \not\leq_{[\mathcal{V}(f\ t_1 \cdots t_n)]} \sigma_2$, since l_1 and l_2 are not unifiable. □

This theorem will be used to prove the termination of the matching algorithm, since it enables the use of the proper non-decreasing property.

2.4.2 Termination

Theorem 2.4 (Termination) Let Γ be a matching problem in which all functions are matching combinators, then every transformation sequence of $\mathcal{TS}^1_d$ terminates. □

The central point of the proof is the definition of an ordering on matching problems. Let t be any term, $|t|$ denotes the depth of defined functions in t and is defined as:

$$\begin{array}{lll} |a| & = 0, & \text{if } a \text{ is a variable or a constant} \\ |c(t_1, \cdots, t_n)| & = \max_{1 \leq i \leq n} |t_i|, & \text{if } c \text{ is a constructor} \\ |f(t_1, \cdots, t_n)| & = \max_{1 \leq i \leq n} |t_i| + 1, & \text{if } f \text{ is a defined function.} \end{array}$$

Let P be a suitable property along with the well-founded partial ordering $\rhd$. The partial ordering on matching pairs $\rhd_p$ is defined as: $(t, s) \rhd_p (t', s')$ iff $P(s) \rhd P(s')$, or $P(s) = P(s')$ and $|t'| < |t|$, where $<$ is the less-than relation on natural numbers. Obviously, $\rhd_p$ is a well-founded partial-ordering.

Now we can further extend the ordering to matching problems. Let $\Gamma \stackrel{rep}{=} \Gamma_0$ & Γ_1 & Γ_2 and $\Gamma' \stackrel{rep}{=} \Gamma'_0$ & Γ'_1 & Γ'_2 be two matching problems, $\Gamma \rhd_\Gamma \Gamma'$ iff $\Gamma_1 \cup \Gamma_2$ $(\rhd_p)_{mul}$ $\Gamma'_1 \cup \Gamma'_2$, where $(\rhd_p)_{mul}$ is the multiset ordering of $\rhd_p$. So $\rhd_\Gamma$ is also well-founded, since $\rhd_p$ is well-founded.

Proof (Theorem 2.4). Let

$$\Gamma_1 \Longrightarrow^*_{\mathcal{TS}^1_d} \Gamma' \Longrightarrow_{(\text{Narrowing})} \Gamma'' \Longrightarrow^*_{\mathcal{TS}^1_d} \Gamma_2$$

be a sequence of transformations using $\mathcal{TS}^1_d$. If we can show that $\Gamma_1 \rhd_\Gamma \Gamma_2$, then the theorem can be proved using the fact that $\rhd_\Gamma$ is a well-founded partial ordering and all other rules can be used to a given matching problem only finitely many times.

Without loss of generality, we assume that Γ_1 contains only one matching pair (t, a) such that

- t has the form $f\ s_1 \cdots s_n$,
- f is an acyclic matching combinator,
- a is a ground term, otherwise we can treat the variables in a as constants, and
- t contains no defined function nested below any proper decreasing function.

$$\begin{array}{ll} & \{\} \ \&\ \{(t, a)\} \ \&\ \{\} \\ \Longrightarrow_{(\text{Abstraction})} & \{\} \ \&\ \{(t[x_1, \cdots, x_k]_{\{p_1, \cdots, p_k\}}, a)\} \ \&\ \{(t|_{p_1}, x_1), \cdots, (t|_{p_k}, x_k)\} \\ \Longrightarrow_{(\text{Narrowing})} & \Gamma_\sigma \ \&\ \{(\sigma r, a)\} \cup (\sigma\Gamma_{2,1}) \ \&\ (\sigma\Gamma_{2,2}), \end{array}$$

where $\Gamma_{2,1} \cup \Gamma_{2,2} = \{(t|_{p_1}, x_1), \cdots, (t|_{p_k}, x_k)\}$, $f\ l_1 \cdots l_n = r$ is the equation used, and $\sigma = mgu(t[x_1, \cdots, x_k]_{\{p_1, \cdots, p_k\}}, f\ l_1 \cdots l_n)$. We show, at first, $(t, a) \rhd_p (\sigma r, a)$. There are two cases to be considered.

- σr has a variable at the top-level. Then only (Variable) can be used to it and transforms it to a solved pair.
- σr has a constructor at the top-level. Then after using (Decomposition), we get a set of matching pairs (r_1, a_1), $\cdots$, (r_m, a_m), where each a_i is a proper subterm of a, so $(\sigma(r), a) \rhd_p (r_i, a_i)$, $1 \le i \le m$.

Thus, by applying the inductive hypothesis we can assume that all matches for $(\sigma r, a)$ can be found in finite time. Let τ be one of these matches, we have:

$$\tau(\sigma(f\ s_1 \cdots s_n)) =_{\mathcal{E}} \tau(\sigma r) =_{\mathcal{E}} a.$$

Then we show that for any $(t|_{p_i}, x_i) \in \Gamma_2$, $(t, a) \rhd_p ((\tau \circ \sigma)t|_{p_i}, (\tau \circ \sigma)x_i)$. The following two cases should be considered:

- Function f is decreasing. By assumption there is no defined function below it, that is, the function abstraction of t is t itself, $\Gamma_2 = \emptyset$.

- Function f is non-decreasing. So, $P(a) \rhd P((\tau \circ \sigma)x_i)$ or $P(a) = P((\tau \circ \sigma)x_i)$, $1 \leq i \leq k$, according to the definition of proper non-decreasing functions and the fact that $\tau \circ \sigma$ is a minimal match such that $\tau(\sigma(f\ t_1 \cdots t_n)) =_{\mathcal{E}} a$. Additionally, $|(\tau \circ \sigma)t|_{p_i}| < |t|$, since each l_i contains no occurrence of any defined function. Therefore, we have $(t, a) \rhd_p ((\tau \circ \sigma)t|_{p_i}, (\tau \circ \sigma)x_i)$. $\square$

Combining these results, we obtain the following theorem for our matching algorithm.

Theorem 2.5 (Finitary) Let Γ be a matching problem in which all functions are matching combinators, then the matching algorithm $\mathcal{MA}^1_d$ terminates with a complete set of minimal matches of Γ. $\square$

Corollary 2.1 Let Γ be a ground matching problem, $\Gamma \Longrightarrow^*_{\mathcal{TS}^1} \Gamma'$ and Γ' is in solved form, then there exists a transformation sequence $\{\}\ \&\ \Gamma\ \&\ \{\} \Longrightarrow^*_{\mathcal{TS}^1_d} \Gamma''\ \&\ \{\}\ \&\ \{\}$, such that $\sigma_{\Gamma''} \leq_{[\mathcal{V}(\Gamma)]} \sigma_{\Gamma'}$. $\square$

2.5 Applications

One major advantage of our approach is the easy combination of constructor-based specifications. The comparison to second-order matching and rewriting modulo commutativity through examples shows the power of our method.

2.5.1 Modularity of Constructor-Based Specifications

One all-pervasive concern in computing science is **modularity**: the ability of composing the solution to a problem from the solutions to its subproblems. Usually, the combinations of unification and matching problems are rather difficult (see [74], [57]); but the combination of matching problems with respect to constructor-based specifications is rather direct.

Disjoint combination

Definition 2.21 (Disjoint combination) Let $\mathcal{SP}_1 = \langle \Sigma_1, \mathcal{E}_1 \rangle$ and $\mathcal{SP}_2 = \langle \Sigma_2, \mathcal{E}_2 \rangle$ be two specifications, and $\Sigma_1 = \langle \mathcal{S}, \mathcal{C}, \mathcal{F}_1 \rangle$, $\Sigma_2 = \langle \mathcal{S}, \mathcal{C}, \mathcal{F}_2 \rangle$. Then the disjoint combination $\mathcal{SP}_1 \oplus \mathcal{SP}_2$ is the specification obtained by renaming $\mathcal{F}_1$ and $\mathcal{F}_2$ such that they are disjoint, then taking the union of their signatures and equations.

The definition of disjoint combination of two term rewriting systems can be given similarly. For example, if $Bool_{(and,or)}$ and $Bool_{(not)}$ are the following specifications, then $Bool_{(and,or)} \oplus Bool_{(not)}$ is the specification $Bool$ with all the three function *and*, *or*, and *not*.

$$
\begin{array}{rl}
Bool_{(and,or)} = & \\
\textbf{sorts}: & bool \\
\textbf{cons}: & true, false \ : \ bool \\
\textbf{funs}: & and,\ or \ : \ bool \to bool \to bool \\
\textbf{eqns}: & \forall\, x \in bool: \\
 & and\ false\ x = false \\
 & and\ true\ x = x \\
 & or\ false\ x = x \\
 & or\ true\ x = true
\end{array}
$$

$$
\begin{array}{rl}
Bool_{(not)} = & \\
\textbf{sorts}: & bool \\
\textbf{cons}: & true, false \ : \ bool \\
\textbf{funs}: & not \ : \ bool \to bool \\
\textbf{eqns}: & \forall\, x \in bool: \\
 & not\ false = true \\
 & not\ true = false
\end{array}
$$

The following theorem is due to Toyama, Klop and Barendregt [82].

Theorem 2.6 Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two left linear term rewriting systems. Then $\mathcal{R}_1 \oplus \mathcal{R}_2$ is convergent iff $\mathcal{R}_1$ and $\mathcal{R}_2$ are convergent.

With this theorem, the following proposition is obvious.

Proposition 2.1 $\mathcal{SP}_1$ and $\mathcal{SP}_2$ are two constructor-based specifications iff their disjoint combination $\mathcal{SP}_1 \oplus \mathcal{SP}_2$ is a constructor-based specification. A function f is a matching combinator in $\mathcal{SP}_1$ or $\mathcal{SP}_2$ iff it is a matching combinator in $\mathcal{SP}_1 \oplus \mathcal{SP}_2$.

Extensions

In section 2.1.1 we defined the extension of a specification $\mathcal{SP}_1$ with $\mathcal{SP}_2$, denoted as $\mathcal{SP}_1 + \mathcal{SP}_2$, which is the specification

$$\langle\langle \mathcal{S}_1 \cup \mathcal{S}_2, \mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{F}_1 \cup \mathcal{F}_2\rangle, \mathcal{E}_1 \cup \mathcal{E}_2\rangle,$$

where $\mathcal{SP}_1 = \langle\langle \mathcal{S}_1, \mathcal{C}_1, \mathcal{F}_1\rangle, \mathcal{E}_1\rangle$ and $\mathcal{SP}_2 = \langle\langle \mathcal{S}_2, \mathcal{C}_2, \mathcal{F}_2\rangle, \mathcal{E}_2\rangle$, $\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset$, $\mathcal{C}_1 \cap \mathcal{C}_2 = \emptyset$, and $\mathcal{F}_1 \cap \mathcal{F}_2 = \emptyset$.

List and *Tree* are both extensions of *Nat*. Different from disjoint combination, the extension of a constructor-based specification need not to be a constructor-based specification. Extension can also change matching combinators, because through extension, some new equations can be introduced, which may destroy the orthogonality of the specification. So, whether an extension is a constructor-based specification and a function in an extended specification is a matching combinator should be checked. *List* and *Tree* are both constructor-based specifications, and all the functions in them are matching combinators.

2.5.2 Example: Contexts and Terms

Contexts are now defined as 'terms' containing one occurrence of a special symbol $\Box$, and are denoted by $C[\,]$. If $\Box$ is substituted by t, the result is $C[t]$; t is said to be a subterm of $C[t]$. The following is a specification. Without loss of generality, we assume that there is only one variable and one constant in *term* and *ctx*.

$$
\begin{array}{lll}
Ctx & = & \\
& \textbf{sorts}: & term,\ ctx \\
& \textbf{cons}: & v, c\ : term \\
& & ap\ : term \to term \to term \\
& & \Box, vv, cc\ : ctx \\
& & ap_c\ : ctx \to ctx \to ctx \\
& \textbf{funs}: & \uparrow\ : ctx \to term \to term \\
& \textbf{eqns}: & \forall\, t \in term;\ c_1, c_2 \in ctx: \\
& & \Box\ \uparrow\ t = t \\
& & vv\ \uparrow\ t = v \\
& & cc\ \uparrow\ t = c \\
& & (ap_c\ c_1\ c_2)\ \uparrow\ t = ap\ (c_1 \uparrow t)\ (c_2 \uparrow t)
\end{array}
$$

It is not difficult to see that *Ctx* is a constructor-based specification and $\uparrow$ is an acyclic matching combinator. There are some matching problems associated with contexts, such as "split s into a context C with a subterm t", i.e., $\{(c \uparrow t, s)\}$ with c a variable, t and

s ground terms; and "find all contexts and subterms of a given term s", i.e., $\{(c \uparrow x, s)\}$ with c and x variables, s a ground term. Solving this kind of matching problems is very important in program transformations. Furthermore, *Ctx* can be generalized to the case that a context contains a list of holes and the function $\uparrow$ substitutes these holes with a list of terms simultaneously.

Using the matching algorithm, the matching problem $\{(x \uparrow y,\ ap\ c\ v)\}$ has the following solutions $\{x \mapsto ap_c\ cc\ \Box,\ y \mapsto vv\}$ or $\{x \mapsto ap_c\ cc\ vv\}$.

Comparing the function $\uparrow$ to general function applications, we can find that matching problems with $\uparrow$ simulate some second-order matching problems. In the second-order case, the matching problem $\{(x\ y, c\ v)\}$ has the solutions $\{x \mapsto \lambda z.c\ z,\ y \mapsto v\}$ and $\{x \mapsto \lambda z.c\ v\}$, see chapter 4.

2.5.3 Example: Representation of Transformation Rules

At first we give two new specifications: *PermSel* for functions *perm* and *sel*, and *Opt* for *option*.

$$
\begin{array}{rrl}
PermSel[\alpha] & = & Bool\ +\ Pair[\alpha,\alpha]\ + \\
 & \textbf{funs}: & perm\ :\ bool \rightarrow pair \rightarrow pair \\
 & & sel\ :\ bool \rightarrow pair \rightarrow \alpha \\
 & \textbf{eqns}: & \forall\ (x,y) \in pair: \\
 & & perm\ true\ (x,y) = (x,y) \\
 & & perm\ false\ (x,y) = (y,x) \\
 & & sel\ true\ (x,y) = x \\
 & & sel\ false\ (x,y) = y
\end{array}
$$

$$
\begin{array}{rrl}
Opt[\alpha] = & & \\
 & \textbf{sorts}: & opt \\
 & \textbf{cons}: & some\ :\ \alpha \rightarrow opt \\
 & & none\ :\ opt \\
 & \textbf{funs}: & option\ :\ bool \rightarrow \alpha \rightarrow opt \\
 & \textbf{eqns}: & \forall\ x \in \alpha;\ o \in opt: \\
 & & option\ true\ x = some\ x \\
 & & option\ false\ x = none
\end{array}
$$

With *perm* and *sel*, the following four program transformation rules about the Boolean functions $\vee$ and $\wedge$:

$$
\begin{array}{lcll}
\wedge(\vee(X,Y),Z) & \Longrightarrow & \vee(\wedge(X,Z),\wedge(Y,Z)) & (1) \\
\wedge(Z,\vee(X,Y)) & \Longrightarrow & \vee(\wedge(Z,X),\wedge(Z,Y)) & (2) \\
\vee(\wedge(X,Y),Z) & \Longrightarrow & \wedge(\vee(X,Z),\vee(Y,Z)) & (3) \\
\vee(Z,\wedge(X,Y)) & \Longrightarrow & \wedge(\vee(Z,X),\vee(Z,Y)) & (4)
\end{array}
$$

can be abstracted by only one transformation rule:

$$
\begin{array}{ll}
 & (sel\ D\ (\wedge,\vee))(perm\ B\ ((sel\ D\ (\vee,\wedge))(X,Y),Z)) \\
\Longrightarrow & (sel\ D\ (\vee,\wedge)) \\
 & ((sel\ D\ (\wedge,\vee))(perm\ B\ (X,Z)),(sel\ D\ (\wedge,\vee))(perm\ B\ (Y,Z))).
\end{array}
$$

If both D and B are *true*, it is rule (1); if D is *true* but B is *false*, it is rule (2); if D is *false* but B is *true*, it is rule (3); and if both D and B are *false*, it is rule (4). Obviously, this kind of representations of transformation rules makes program transformations very abstract. Readability could be improved by some syntactic sugar. For instance, to transform the term

$$\wedge(\vee(b_1, b_2), b_3),$$

instead of trying to match the term with each left-hand side of the four rules, we only need to solve the following matching problem:

$$\{(sel\ D\ (\wedge, \vee))(perm\ B\ ((sel\ D\ (\vee, \wedge))(X, Y), Z)), \wedge(\vee(b_1, b_2), b_3))\}.$$

The only match is

$$\{D \mapsto true, B \mapsto true, X \mapsto b_1, Y \mapsto b_2, Z \mapsto b_3\}.$$

With matching combinator *option* we can easily present an "IF-THEN-ELSE" expression with an optional "ELSE" part:

$$\text{IF } x_c \text{ THEN } s_1\ (option\ x_b \text{ ELSE } s_2).$$

An interesting application of *perm* is to present rewriting modulo commutativity. Let R be a set of rewriting rules:

$$\begin{array}{lcl} x + 0 & \rightarrow & x \\ x + (-x) & \rightarrow & 0 \end{array}$$

In rewriting modulo commutativity (see [65], [43], also section 2.3.3), the system R_C contains all rules $s \rightarrow t$, such that $s =_C \sigma(s')$ and $t = \sigma(t')$, for some rule $s' \rightarrow t'$ in R. Using *perm* we can extend R to R':

$$\begin{array}{lcl} +(perm\ x_b\ (x, 0)) & \rightarrow & x \\ +(perm\ x_b\ (x, -x)) & \rightarrow & 0 \end{array}$$

It is not difficult to see that R' is equivalent to R_C. It seems that quite a large class of rewriting modulo a congruence can be simulated by functions defined in constructor-based specifications. We feel that this is another aspect in which constructor-based specifications turn out to be practically useful and theoretically interesting.

2.6 Related Work

Matching problems associated with constructor-based specifications are a special class of $\mathcal{E}$-matching problems, which are particularly interesting for functional programming and program transformation. The comparison to second-order matching and rewriting modulo commutativity shows the power of our method. The easy combination of constructor-based specifications is also an advantage of our approach. The contribution of this chapter is not only the result itself – the decidable first-order semantic matching — but the combination of the following points:

- The class of constructor-based specifications has a complete semantic matching algorithm, which may be implemented efficiently.

- Matching combinators can be restricted to the class of acyclic matching combinators, such that the matching algorithm always terminates.

- A proof technique has been described, which copes with the asymmetric nature of matching problems, particularly in the case of non-ground matching problems introduced by narrowing.

In the rest of this section , we compare our research to related work.

Decidable semantic matching

In [17], Dershowitz, Mitra and Sivakumar have discussed the decidability of semantic matching for certain classes of term rewriting systems. A simple system is provided, which is complete and terminating with respect to these classes of term rewriting systems.

The differences between the left linear rewrite systems in [17], and the constructor-based specifications are the following:

- The concept of non-decreasing functions is generalized to proper non-decreasing functions. It is appropriate for functions containing both variable-preserving and variable-dropping equations.

- We require that the left-hand side of each equation contains only one occurrence of a defined function, i.e., as its top-level symbol.

The system $\mathcal{TS}^1$ presented in this chapter is more constructive than that in [17]. Particularly, we have introduced "structural narrowing" to solve equations during the matching process, instead of using the general "Mutate" rule as in [17].

$$\text{(Mutate)} \quad \{(f\ t_1 \cdots t_n, t)\} \cup \Gamma \Longrightarrow \{(t_1, l_1), \cdots, (t_n, l_n), (r, t)\} \cup \Gamma,$$

where $f\ l_1 \cdots l_n \to r$ is a renamed rule. Structural narrowing is more efficient than (Mutate), because only rules whose left-hand sides are unifiable with the term will be considered. We aim not only at theoretical results, but also at implementation and application.

Structural narrowing and basic narrowing

The central point of the matching algorithm is the introduction of structural narrowing, which is different from the existing narrowing strategies, such as basic narrowing [41], normal narrowing [21], innermost narrowing [22], and lazy narrowing [86]. In our case, structural narrowing helps to find impossible top-level symbols of s in a matching pair (s,t) as early as possible, avoids unnecessary narrowing steps, and makes implementation efficient and easy. We compare here structural narrowing with basic narrowing that has been widely studied.

At first we recall the definition of basic narrowing from [41].

Definition 2.22 (Basic narrowing) Let the following be a narrowing derivation:

$$t_1 \rightsquigarrow_{p_1, l_1 \to r_1} t_2 \rightsquigarrow_{p_2, l_2 \to r_2} \cdots t_{n-1} \rightsquigarrow_{p_{n-1}, l_{n-1} \to r_{n-1}} t_n,$$

where for any i $(1 \leq i \leq n-1)$, the substitution is applied at occurrence p_i. The set of occurrences $U_i \subset \mathcal{O}(t_i)$, $1 \leq i \leq n$, is constructed as follows:

- $U_1 = \overline{\mathcal{O}}(t_1)$,
- $U_{i+1} = (U_i - \{q \in U_i \mid p_i \text{ is above } q\}) \cup \{p_i \cdot q \mid q \in \overline{\mathcal{O}}(r_i)\}$.

The above narrowing derivation is called **basic** if $p_i \in U_i$ for all $1 \leq i \leq n-1$.

In basic narrowing derivation, narrowing is never applied to a subterm introduced by a previous narrowing substitution. The following theorem is due to Hullot [41] and some other authors (Yamamoto [85] and Hölldobler [38]).

Theorem 2.7 Basic narrowing is complete for convergent term rewriting systems. □

According to the theorem, only basic narrowing needs to be considered. To guarantee the termination of a unification or matching algorithm using basic narrowing, we should prove that any basic narrowing derivation terminates. The latter can be proved if each basic narrowing derivation issuing from any r, where $l \to r$ is a rewriting rule, terminates.

In structural narrowing derivation, we just apply narrowing to a term as a whole, i.e., always at the occurrence Λ. It is obviously simpler than basic narrowing. In addition, we have proved that structural narrowing together with combinator abstraction is complete for constructor-based specifications. So we only need to consider structural narrowing. The termination of any structural narrowing derivation is ensured by the definition of acyclic matching combinators.

Structural narrowing and outermost narrowing

It is not difficult to see that structural narrowing is in fact a lazy narrowing strategy, since it delays the evaluation of function arguments until their values are definitely needed to compute the result of the function call. **Outermost narrowing** is another lazy narrowing strategy, where the next narrowing position should be outermost. Outermost narrowing is incomplete as the following example shows [19]:

$$\begin{array}{lcl} f\ 0\ 0 & = & 0 \\ f\ (succ\ x)\ 0 & = & succ\ 0 \\ f\ x\ (succ\ y) & = & succ(succ\ 0). \end{array}$$

We want to solve the matching problem $\{(f\ (f\ i\ j)\ k, 0)\}$. Obviously, $\{i \mapsto 0,\ j \mapsto 0,\ k \mapsto 0\}$ is a match. However, by using outermost narrowing, only the third rule can be applied, and no solution will be found.

To solve the matching pair $\{(f\ (f\ i\ j)\ k, 0)\}$, before applying structural narrowing, we first abstract the term $f\ (f\ i\ j)\ k$, and get a new matching problem $\{(f\ z\ k, 0), (f\ i\ j, z)\}$. Solving $(f\ z\ k, 0)$ we have the substitution $\{z \mapsto 0,\ k \mapsto 0\}$. Now we can solve $(f\ i\ j, 0)$ and get the substitution $\{i \mapsto 0,\ j \mapsto 0\}$. So the match for the original problem is $\{i \mapsto 0,\ j \mapsto 0,\ k \mapsto 0\}$.

Echahed in [19] and Padawitz in [62] have investigated conditions for the completeness of arbitrary narrowing strategies. However, their most interesting applications are outermost strategies. In addition to confluence and termination of the set of equations, complete narrowing strategies must satisfy a **uniformity** condition. Uniformity means that the position selected by the narrowing strategy is a valid narrowing position for all substitutions in normal form applied to it. The outermost strategy is not uniform in general since, in the last example, the top position of the term $f\ (f\ i\ j)\ k$ is not a valid position if we apply the substitution $\{k \mapsto 0\}$ to this term. Structural narrowing satisfies the uniformity condition.

Martelli and Montanari's unification algorithm

The dependency relation on matching pairs is motivated by the partial ordering on multiequations of first-order unification problems from Martelli and Montanari, but technically they are quite different. In [52] Martelli and Montanari put forward an efficient first-order unification algorithm for specifications without equations. The central points of the algorithm are the following:

- the representation of unification problems as a set of multiequations of the form
$$\{x_1, \cdots, x_n\} = (t_1, \cdots, t_m);$$
- the introduction of a partial ordering on multiequations: $S_1 < S_2$ iff there exists a variable on the left-hand side of S_1, which also occurs in the right-hand side of S_2.

For instance, the unification problem

$$\{(f\ x_1\ (g\ x_2\ x_3), f\ (g\ (h\ a\ x_4)\ x_2)\ x_1)\}$$

can be represented as the following set of multiequations;

$$\begin{array}{lrcl} \{ & \{x\} & = & (f\ x_1\ (g\ x_2\ x_3), f\ (g\ (h\ a\ x_4)\ x_2)\ x_1) \\ & \{x_1\} & = & \emptyset \\ & \{x_2\} & = & \emptyset \\ & \{x_3\} & = & \emptyset \\ & \{x_4\} & = & \emptyset \qquad \} \end{array}$$

The following theorem (see [52]) ensures that any set of multiequations can be solved by choosing a multiequation S, such that there is no S' satisfying $S < S'$.

Theorem 2.8 If a set of multiequations has a unifier, then the transitive closure $<^*$ of $<$ is a partial ordering. $\square$

The first multiequation in the above example should be chosen first. Solving it we get a new set of multiequations

$$\begin{array}{llll} \{ & \{x\} & = & (f\ x_1\ x_1) \\ & \{x_1\} & = & (g\ x_2\ x_3, g\ (h\ a\ x_4)\ x_2) \\ & \{x_2\} & = & \emptyset \\ & \{x_3\} & = & \emptyset \\ & \{x_4\} & = & \emptyset \qquad\qquad\qquad\qquad \} \end{array}$$

Now the second one should be chosen, and the process can be continued.

After taking an ordering on multiequations, the solution is kept in a factorized form and no substitution ever has to be applied, so the efficiency of first-order unification has been improved.

It is not difficult to see that this process does not help to solve matching problems. To construct the set of multiequations of any matching problem is in fact to solve the problem, since one term in a matching pair is ground. In addition, all variables in multiequations, except one introduced at the beginning (e.g., x in the above example), are in original problems. Dependency relations between matching pairs will be changed dynamically, and new variables will be introduced. At the beginning, there is no dependency relation between the matching pairs of a ground matching problem. After applying rule (Abstraction), they will be introduced. The (Narrowing) transformation may introduce some new matching pairs that can be abstracted by (Abstraction). Using dependency analysis, useless narrowing steps are avoided. Variables will be substituted only once, and matching trees are much easier to maintain. The efficiency of the matching algorithm is improved.

This is to say, both, the partial ordering on multiequations and the dependency relation on matching pairs, are used to improve the efficiency of a unification or matching algorithm by choosing a "best subgoal" to be solved first. Nevertheless, they are introduced for different considerations, so the ways to use them and the methods to compute them are different.

Chapter 3

Contextual Matching

The problem of contextual matching comes from conditional term rewriting systems or specifications with conditional equations. Let the following be a rewrite rule of a term rewriting system:

$$x ++ y \;\rightarrow\; x \;\Leftarrow\; x = rev\, y.$$

The rule can only be applied to rewrite a term if the condition $x = rev\, y$ is satisfied, i.e., x is the reverse list of y. For instance, to rewrite the palindrome list $1 :: (2 :: (2 :: (1 :: nil)))$ using this rule, we should solve the matching problem:

$$\{(x ++ y, 1 :: (2 :: (2 :: (1 :: nil))))\}\ [x = rev\, y].$$

This means to find all possible matches σ of the matching problem $\{(x ++ y,\ 1 :: (2 :: (2 :: (1 :: nil))))\}$ such that $\sigma x =_{\mathcal{E}} rev(\sigma y)$, where $\mathcal{E}$ is a set of equations defining the function *rev*.

The name "contextual matching" is derived from the "contextual rewriting" discussed by Zhang, Remy and Kapur in [89, 88, 87]. Generally, contextual matching should solve matching problems with extra conditions — contexts. To solve a matching pair (s, t) with context C, in addition to finding a set of matches of (s, t), we also have to prove whether the context C is satisfiable under these matches. Our goal is to reduce the number of matches and to speed up the matching process through using context information. As mentioned, there are 651 matches of the matching problem "all the trees of height 4". If the matching problem is changed to "all balanced trees of height 4", then there is only one match, where "balanced trees" can be seen as a context.

In this chapter our interest is more on techniques to solve contextual matching problems than on the theoretical results. Most of the work about contextual rewriting or contextual narrowing – term rewriting or narrowing with context information – is also motivated by practical problems and concerns for the solutions of technical problems, see [24], [88], [89] and [87]. We will focus on the following two points:

- how to define the contexts of matching problems, and
- how to use and to verify contexts during the matching process.

3.1 Context

Generally, there are two methods to cope with conditional term rewriting. To rewrite a term t with a conditional rule $l \rightarrow r \Leftarrow C$, we simply match the left-hand side l with t at first, and get some substitutions if they are matchable, then prove condition C under these substitutions. t can be rewritten if C is satisfiable. Instead of "matching then prove", "prove during matching" is also a possible way. That is to say, the condition C will be proved during matching l with t, and we get some substitutions if they are matchable under C, then t can be rewritten. The former is simple and easy to implement; but the latter may be more efficient, particularly if semantic matching will be considered. The introduction of contexts will reduce the number of matches, thus speeding up the matching process.

Again, there are two methods to treat contexts during a matching process: contexts as additional matching problems, or contexts as additional conditions. We will treat contexts as additional conditions; the reason can be found in section 3.3 of this chapter. Variables in contexts will be gradually instantiated, and they will be simplified through term rewriting.

Under these considerations, we give the following definition.

Definition 3.1 (Context) Let $\mathcal{SP}$ be a convergent specification. A **context equation** $EQ(t,s)$ or **context disequation** $NEQ(t,s)$ is a pair of terms, where t and s are two terms of the same sort from $\mathcal{SP}$. A context C is a set of context equations and disequations, denoted as $[e_1, \cdots, e_m]$.

Remember that functions in any matching problem should be acyclic matching combinators to ensure the termination of the matching process. To make contexts computable, we only require that all functions are defined by convergent specifications, since contexts will be simplified through term rewriting, so the condition is relaxed in contexts. The context equations and disequations can also be solved through unification or disunification. In [11], Comon summarized the research about solving equations and disequations together, and called it "equational problem" solving. This is out of our area.

$EQ(\mathit{isBal}\ x, \mathit{true})$ is a context equation, where isBal is defined by the following specification:

$$
\begin{array}{rll}
\mathit{TreeB} & = & \mathit{Tree}\ +\ \mathit{Bool}\ + \\
 & \textbf{funs}: & \mathit{eqNat}\ :\ \mathit{nat} \to \mathit{nat} \to \mathit{nat} \\
 & & \mathit{isBal}\ :\ \mathit{tree} \to \mathit{bool} \\
 & \textbf{eqns}: & \forall\ x, y \in \mathit{nat};\ \ xt, yt \in \mathit{tree}: \\
 & & \mathit{eqNat}\ 0\ 0 = \mathit{true} \\
 & & \mathit{eqNat}\ (\mathit{succ}\ x)\ 0 = \mathit{false} \\
 & & \mathit{eqNat}\ (\mathit{succ}\ x)\ (\mathit{succ}\ y) = \mathit{eqNat}\ x\ y \\
 & & \\
 & & \mathit{isBal}\ \mathit{empty} = \mathit{true} \\
 & & \mathit{isBal}(\mathit{node}\ xt\ yt) = \mathit{and}\ (\mathit{eqNat}\ (\mathit{height}\ xt)\ (\mathit{height}\ yt)) \\
 & & \qquad\qquad\qquad\qquad\quad (\mathit{and}\ (\mathit{isBal}\ xt)\ (\mathit{isBal}\ yt))
\end{array}
$$

Note that both eqNat and isBal are not acyclic matching combinators, because the right-hand sides of the third and the last equation begin with function symbols. Furthermore, it is impossible to transform them to acyclic matching combinators.

Convention:

- If e is a context equation $EQ(t,s)$ or disequation $NEQ(t,s)$, then $\mathcal{V}(e) = \mathcal{V}(t) \cup \mathcal{V}(s)$; if $C = [e_1, \cdots, e_m]$, then $\mathcal{V}(C) = \mathcal{V}(e_1) \cup \cdots \cup \mathcal{V}(e_m)$.

- If e is a context equation $EQ(t, s)$ or disequation $NEQ(t, s)$ and σ is a substitution, then σe is the context equation $EQ(\sigma t, \sigma s)$ or $NEQ(\sigma t, \sigma s)$; if $C = [e_1, \cdots, e_m]$, then $\sigma C = [\sigma e_1, \cdots, \sigma e_m]$.

Definition 3.2 (Contextual matching problems) A contextual matching problem is a matching problem Γ with a context C, denoted as $\Gamma\ C$.

The matching problem "all balanced trees of height 4" can now be presented as:

$$\{(height\ (node\ t_1\ t_2), succ(succ(succ(succ\ 0))))\}\ [EQ(isBal(node\ t_1\ t_2), true)].$$

This definition is due to the following considerations:

- To avoid redundant presentation, we do not distribute the context of a matching problem to its matching pairs. See the following example:

$$\{(x +\!\!+ (y +\!\!+ z), t)\} \Longrightarrow_{\text{(Abstraction)}} \{(x +\!\!+ x', t), (y +\!\!+ z, x')\},$$

  the matching pair $(x +\!\!+ (y +\!\!+ z), t)$ is abstracted by two new pairs. If each matching pair is associated with a context, then after such transformations, the context may appear several times. Consequently, the context will be computed more than once.

- If $\Gamma\ C$ is a contextual matching problem, we do not require $\mathcal{V}(C) \subseteq \mathcal{V}(\Gamma)$, i.e., C can contain variables that do not occur in Γ. We know that if σ is a match of Γ, σx may be a non-ground term for some $x \in \mathcal{D}(\sigma)$. Additionally, $\mathcal{V}(\Gamma)$ is not necessarily equal to $\mathcal{D}(\sigma)$. See the following contextual matching problem:

$$\{(len(y :: ys), succ(succ\ 0))\}\ [EQ(y, 0)];$$

  the only minimal match of the matching problem is $\{ys \mapsto y_1 :: nil\}$, which contains no y and is a non-ground substitution as well.

The computation of contexts consists of the three basic steps:

- Simplification through term rewriting.
- Instantiation through substitution.
- Verification through constant equivalence and non-equivalence, which will be discussed below.

Definition 3.3 Let t and s be two normal forms. t and s are called **constant equivalent**, denoted as $t \stackrel{c}{=} s$, iff

- t and s are the same constant; or
- $t = f\ t_1 \cdots t_n$, $s = f\ s_1 \cdots\ s_n$, where f is an n-ary ($n > 0$) constructor or function symbol, and $t_i \stackrel{c}{=} s_i$ for all $1 \leq i \leq n$.

t and s are called **constant non-equivalent**, denoted as $t \stackrel{c}{\neq} s$, iff

- t and s are ground terms but t and s are not constant equivalent; or

- $t = c_1\, t_1 \cdots\, t_n$, $s = c_2\, s_1 \cdots\, s_n$, and c_1 and c_2 are different constructor symbols.

In constant equivalence or non-equivalence, only constants can be compared, variables are not comparable. Ground normal forms can be constant equivalent, for instant, $succ\ 0 \stackrel{c}{=} succ\ 0$, but $succ\ x \stackrel{c}{=} succ\ 0$ does not hold, since x is a variable. Constant non-equivalence is different from non-unifiability with respect to empty set of equations. If $t \stackrel{c}{\neq} s$, then s and t are not unifiable, but not vice versa. For instance, $succ\ x \stackrel{c}{\neq} 0$, $succ\ x$ and 0 are also not unifiable; x and $succ\ x$ are not unifiable, but $x \stackrel{c}{\neq} succ\ x$ does not hold. Because of the computational difficulty and complexity, equations and disequations in a context will not be solved by unification and disunification. It is possible that two terms are neither constant equivalent nor constant non-equivalent.

Definition 3.4
- $EQ(t, s)$ or $NEQ(t, s)$ is called **satisfiable** iff $t \stackrel{c}{=} s$ and $t \stackrel{c}{\neq} s$ respectively.
- $EQ(t, s)$ or $NEQ(t, s)$ is called **non-satisfiable** iff $t \stackrel{c}{\neq} s$ and $t \stackrel{c}{=} s$ respectively.
- Otherwise, $EQ(t, s)$ or $NEQ(t, s)$ is called **suspended**.
- A context C is satisfiable iff all its context equations and disequations are satisfiable.
- A context C is non-satisfiable iff one of its context equation or disequation is non-satisfiable.
- Otherwise, C is called **suspended**.

For instance, $EQ(succ\ x, 0)$, $NEQ(succ\ 0, succ\ 0)$ are non-satisfiable; $EQ(succ\ 0, succ\ 0)$, $NEQ(succ\ x, 0)$ are satisfiable; $EQ(succ\ x, succ\ 0)$ is suspended.

Definition 3.5 Let $\Gamma\ C$ be a contextual matching problem.

- σ is said to be a **satisfiable match** of $\Gamma\ C$ if σ is a match of Γ and σC is satisfiable.
- σ is said to be a **suspended match** of $\Gamma\ C$ if σ is a match of Γ and σC is suspended.
- σ is said to be a **match** of $\Gamma\ C$ if it is either a satisfiable match or a suspended match.

Not only the satisfiable matches of a contextual matching problem are important, but sometimes its suspended matches can also be meaningful. We can see this from the following example:

$$\begin{array}{ll} & \{(len\ ys, succ(succ\ 0))\}\ [EQ(hd\ ys, 0)] \\ \Longrightarrow_{(\text{Narrowing})} & \{(ys, y_1 :: ys_1), (len\ ys_1, succ\ 0)\}\ [EQ(hd(y_1 :: ys_1), 0)] \\ & \cdots \\ \Longrightarrow^* & \{(ys, (y_1 :: y_2 :: nil)), (ys_1, y_2 :: nil)\}\ [EQ(y_1, 0)] \end{array}$$

Obviously, $\{ys \mapsto y_1 :: (y_2 :: nil)\}$ is a suspended match, but we can not throw it away. The original matching problem says “find a list of length 2 such that its first element is 0”. The match is the substitution $\{ys \mapsto y_1 :: (y_2 :: nil)\}$ together with the context $[EQ(y_1, 0)]$. However, in some practical applications, particularly in program transformations, suspended contexts can be deleted, see examples in section 3.3 of this chapter.

3.2 The Matching Algorithm $\mathcal{MA}^C$

Let Γ be a matching problem and C a context. We assume that every term in Γ and in C is automatically normalized.

The **system** $\mathcal{TS}^C$ for solving contextual matching problems consists of nine rules. It begins with a contextual matching problem $\Gamma\ C$ such that Γ is a ground matching problem and $\mathcal{V}(C) \subseteq \mathcal{V}(\Gamma)$, and terminates with a contextual matching problem $\Gamma'\ C'$ such that Γ' is in solved form and C' is empty or suspended. $\mathcal{TS}^C$ is an extension of the system $\mathcal{TS}^1$ presented in section 2.2. The rules (Remove), (Deletion), (Decomposition) and (Abstraction) remain the same, while in the rules (Variable) and (Narrowing), the substitution will also be applied to the context, so that the context becomes more concrete.

(Variable) finds a partial solution for a variable:

$$(\{(x,t)\} \cup \Gamma)\ C \Longrightarrow (\{(x,t)\} \cup (\sigma\Gamma))\ (\sigma C),$$

if $x \in \mathcal{V}(\Gamma)$, where $\sigma = \{x \mapsto t\}$.

(Narrowing) realizes structural narrowing of a pair:

$$\{(f\ t_1 \cdots\ t_n, s)\} \cup \Gamma)\ C \Longrightarrow (\Gamma_\sigma \cup \{(\sigma r, s)\} \cup (\sigma\Gamma))\ (\sigma C),$$

where $l = r \in \mathcal{E}$, $\sigma = mgu(f\ t_1 \cdots t_n, l)$, and $\mathcal{I}(\sigma) \cap (\mathcal{V}(s) \cup \mathcal{V}(\Gamma)) = \emptyset$.

The following two rules are new to cope with contexts. As defined in section 2.3, *failure* denotes the matching problems that are not matchable.

(Failure) terminates with non-satisfiable context:

$$\Gamma\ [e_1, \cdots, e_m] \implies \textit{failure},$$

if e_i $(1 \le i \le m)$ is non-satisfiable.

(Satisfaction) deletes a satisfiable context equation or disequation:

$$\Gamma\ [e_1, \cdots, e_m] \implies \Gamma\ [e_1, \cdots, e_{i-1}, e_{i+1}, e_m],$$

if e_i $(1 \le i \le m)$ is satisfiable.

Their meanings are obvious. If σ is a match of a matching problem Γ with non-satisfiable context C, then σ is neither a satisfiable match nor a suspended match of $\Gamma\ C$, i.e., σ is not a match of $\Gamma\ C$. The following theorem can be proved using the fact that $\mathcal{TS}^1$ is a sound and complete system.

Theorem 3.1 Let Γ be a ground matching problem, C a context and

$$\Gamma\ C \implies^*_{\mathcal{TS}^C} \Gamma'\ C'$$

be a transformation sequence of the system $\mathcal{TS}^C$, where Γ' is in solved form.
$C' = [\,]$ iff $\sigma_{\Gamma'}$ is a satisfiable match of $\Gamma\ C$.
C' is a suspended context iff $\sigma_{\Gamma'}$ is a suspended match of $\Gamma\ C$. □

We will show some examples. At first we define a new specification *ListR*.

$$
\begin{array}{rll}
ListR & = & List[\alpha] + \\
& \textbf{funs}: & rev \ : \ list \rightarrow list \\
& \textbf{eqns}: & \forall \ x \in \alpha; \ xs \in list : \\
& & rev \ nil = nil \\
& & rev(x :: xs) = (rev \ xs) +\!\!+ (x :: nil)
\end{array}
$$

Obviously, the matching problem $\{(x +\!\!+ y, 1 :: (2 :: (2 :: (1 :: nil))))\} \ [EQ(x, rev \ y)]$ has only one match $\{x \mapsto 1 :: (2 :: nil), \ y \mapsto 2 :: (1 :: nil)\}$. We also have the following transformation sequences:

$$
\begin{array}{ll}
& \{(x +\!\!+ y, 1 :: (2 :: (2 :: (1 :: nil))))\} \ [EQ(x, rev \ y)] \\
\Longrightarrow_{(\text{Narrowing})} & \{(x, nil), (y, 1 :: (2 :: (2 :: (1 :: nil))))\} \ [EQ(nil, rev \ y)] \\
\Longrightarrow_{(\text{Variable})} & \{(x, nil), (y, 1 :: (2 :: (2 :: (1 :: nil))))) \\
& \qquad [EQ(nil, 1 :: (2 :: (2 :: (1 :: nil))))] \\
\Longrightarrow_{(\text{Failure})} & failure
\end{array}
$$

$$
\begin{array}{ll}
& \{(x +\!\!+ y, 1 :: (2 :: (2 :: (1 :: nil))))\}[EQ(x, rev \ y)] \\
\Longrightarrow_{(\text{Narrowing})} & \{(x, x_1 :: xs_1), (x_1 :: (xs_1 +\!\!+ y), 1 :: (2 :: (2 :: (1 :: nil))))\} \\
& [EQ(x_1 :: xs_1, rev \ y)] \\
\Longrightarrow_{(\text{Decomposition})} & \{(x, 1 :: xs_1), (x_1, 1), (xs_1 +\!\!+ y, 2 :: (2 :: (1 :: nil)))\} \\
& [EQ(1 :: xs_1, rev \ y)] \\
\Longrightarrow_{(\text{Narrowing})} & \{(x, 1 :: (x_2 :: xs_2)), (x_1, 1), (xs_1, x_2 :: xs_2), \\
& (x_2 :: (xs_2 +\!\!+ y), 2 :: (2 :: (1 :: nil)))\} \ [EQ(1 :: (x_2 :: xs_2), rev \ y)] \\
\Longrightarrow_{(\text{Decomposition})} & \{(x, 1 :: (2 :: xs_2)), (x_1, 1), (xs_1, 2 :: xs_2), (x_2, 2), \\
& (xs_2 +\!\!+ y, 2 :: (1 :: nil))\} \ [EQ(1 :: (2 :: xs_2), rev \ y)] \\
\Longrightarrow_{(\text{Narrowing})} & \{(x, 1 :: (2 :: nil)), (x_1, 1), (xs_1, 2 :: nil), (x_2, 2), (xs_2, nil), \\
& (y, 2 :: (1 :: nil))\} \ [EQ(1 :: (2 :: nil), rev \ y)] \\
\Longrightarrow_{(\text{Variable})} & \{(x, 1 :: (2 :: nil)), (y, 2 :: (1 :: nil)), (x_1, 1), (xs_1, 2 :: nil), (x_2, 2), \\
& (xs_2, nil)\} \ [EQ(1 :: (2 :: nil), rev(2 :: (1 :: nil)))] \\
\Longrightarrow_{(\text{Satisfaction})} & \{(x, 1 :: (2 :: nil)), (y, 2 :: (1 :: nil)), (x_1, 1), (xs_1, 2 :: nil), \\
& (x_2, 2), (xs_2, nil)\} \ [\ ]
\end{array}
$$

The matching algorithm $\mathcal{MA}^C$ is built upon $\mathcal{MA}^1$ by adding a new module CONTEXT. Figure 8 shows its structure. The main difference between $\mathcal{MA}^1$ and $\mathcal{MA}^C$ is that the context of a matching problem will be rewritten and verified during simplification by the new module CONTEXT. The following is a brief description of CONTEXT.

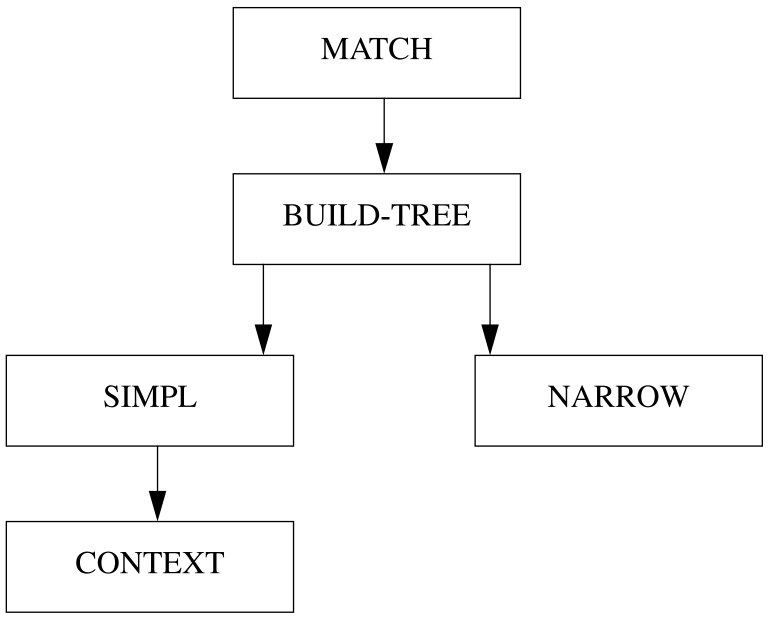

Figure 8: The structure of $\mathcal{MA}^{C}$

3.2.

$\mathsf{CONTEXT}(C)$:
if $C = [\,]$, then $[\,]$
elseif $C = [e] \cup C'$
    then let $e' = \mathit{Rewriting}(e)$ in
        if *Satisfiable?*(e') then $\mathsf{CONTEXT}(C')$
        elseif *Non-satisfiable?*(e') then *failure*
        else $[e'] \cup \mathsf{CONTEXT}(C')$
end $\mathsf{CONTEXT}$.

Rewriting(e) means rewriting two terms of a context equation or disequation e to their normal form. *Satisfiable?* and *Non-satisfiable?* are two functions to prove whether a context equation or disequation is satisfiable or non-satisfiable.

The techniques applied to improve the algorithm $\mathcal{MA}^1$ can also be used to improve $\mathcal{MA}^C$, particularly dependency analysis. In the system $\mathcal{TS}^C$, the context will always be instantiated by (Variable) and (Narrowing). So variables in a context may be substituted several times. After dependency analysis, we only need to substitute variables after solving each solvable matching pair, so that each variable in the context will be substituted only once.

3.3 Discussions and Applications

3.3.1 A Variant Definition

The contextual matching discussed in the previous sections is strongly influenced by the consideration of program transformations. There is another possibility to define contexts of matching problems – contexts as new matching problems.

Definition 3.6 (Contextual matching problems – variation) Let $\mathcal{SP}$ be a specification. Γ $[EQ(t_1, s_1), \cdots, EQ(t_m, s_m)]$ is a contextual matching problem of $\mathcal{SP}$, if both Γ and $\{(t_1, s_1), \cdots, (t_m, s_m)\}$ are matching problems of $\mathcal{SP}$.
σ is said to be a match of Γ C if σ is a match of Γ and a match of $\{(t_1, s_1), \cdots, (t_m, s_m)\}$ as well.

Now the transformation system $\mathcal{TS}^C$ begins with a contextual matching problem Γ $[EQ(t_1, s_1), \cdots, EQ(t_m, s_m)]$, such that both Γ and $\{(t_1, s_1), \cdots, (t_m, s_m)\}$ are ground matching problems; it terminates with Γ' $[EQ(t'_1, s'_1), \cdots, EQ(t'_k, s'_k)]$, such that Γ' is in solved form, and each $EQ(t'_i, s'_i)$ is a suspended context equation, $1 \leq i \leq k$. Then the matching problem $\{(t'_1, s'_1), \cdots, (t'_k, s'_k)\}$ without context can be solved by $\mathcal{TS}^1$.

The differences of these two definitions are the following:

- In the variation, contexts are restricted to matching problems. That is, in a context equation $EQ(t, s)$, s is always a ground term, so equations between variables in a matching problem can not be used as context, such as $EQ(x, rev\ y)$. To solve this problem, we should introduce an "equation function" eq for each sort, then $EQ(x, rev\,y)$ can be presented as $(eq(x, rev\,y), true)$; but eq is not an acyclic matching combinator in general, such as *eqNat* in the above example.

- An important point is that in the original contextual matching we only require that functions in contexts are defined by convergent specifications, instead of only acyclic matching combinators. In the variant definition, suspended contexts will be solved by the matching algorithm. Including non-acyclic matching combinators may cause the matching algorithm to diverge. So functions like *isBal* and *eqNat* are not allowed in the variant definition.

3.3.2 Examples

The following examples demonstrate the practical advantages of the contextual matching introduced in the previous sections.

Lift common part from an "if-then-else" expression

The following is a program transformation rule to optimize an "IF-THEN-ELSE" statement. The common part of both "THEN" and "ELSE" statements will be lifted to the outside of the "IF-THEN-ELSE" statement.

$$\text{IF } x_c \text{ THEN } (s +\!\!+ s_1) \text{ ELSE } (s +\!\!+ s_2) \Longrightarrow s +\!\!+ (\text{IF } x_c \text{ THEN } s_1 \text{ ELSE } s_2).$$

The condition of this transformation rule is that the first statements of both s_1 and s_2 should not be equal, otherwise they also belong to the common part, i.e.,

$$hd\ s_1 \neq hd\ s_2.$$

Let the following be a segment of a program to be transformed

$$\text{IF } f \text{ THEN } [a, b, c, d] \text{ ELSE } [a, b, e],$$

where f is an expression, a, b, c, d and e are all concrete statements, $[a, b, c, d]$ and $[a, b, e]$ denote lists of statements.

After a step of (Decomposition), the matching problem is transformed to:

$$\{(x_c, f), (s +\!\!+ s_1, [a, b, c, d]), (s +\!\!+ s_2, [a, b, e])\}\ [NEQ(hd\ s_1, hd\ s_2)]$$

Solving the matching pair $(s +\!\!+ s_2, [a, b, e])$, we get the following matches:

$$\begin{array}{lll} \{s \mapsto [\,], & s_2 \mapsto [a, b, e]\} & (1) \\ \{s \mapsto [a], & s_2 \mapsto [b, e]\} & (2) \\ \{s \mapsto [a, b], & s_2 \mapsto [e]\} & (3) \\ \{s \mapsto [a, b, e], & s_2 \mapsto [\,]\} & (4) \end{array}$$

Now we solve the matching pair $(s +\!\!+ s_1, [a, b, c, d])$. In the case of (1), $s_1 \mapsto [a, b, c, d]$, but $hd\ s_1 = hd\ s_2 = a$; in the case of (2), $s_1 \mapsto [b, c, d]$, but $hd\ s_1 = hd\ s_2 = b$; in the case of (3), $s_1 \mapsto [c, d]$, and $hd\ s_1 = c \neq e = hd\ s_2$; in the last case, the pair $(s +\!\!+ s_1, [a, b, c, d])$ has no solution.

So the program segment can only be transformed to

$$[a, b,\ (\text{IF } f \text{ THEN } [c, d] \text{ ELSE } [e])].$$

Transformation an "if-then-else" expression to a "case" expression

When solving a contextual matching problem, some suspended matches can also be produced together with its satisfiable matches. This situation occurs rarely if we consider the application in program transformations, because suspended matches occur only if some variables in the original matching problem are substituted by non-ground terms or are not substituted at all. That is,

- a program segment is transformed to a new segment containing an unknown part that can be substituted by any program without changing the meaning of the original program, which generally violates the concept of correctness-preserving transformations; or
- a program may be transformed to a new segment with an extra suspended context that contains variables playing no role in the program, which means that the extra suspended context is unimportant for the program and can be deleted.

The following is an example of a suspended context. The problem comes from the fact that an "IF-THEN-ELSE" statement with optional "ELSE" part can be transformed to a "CASE" statement. *option* is a function defined by the specification $Opt[s]$ in section 2.5. The program transformation rule is

$$\begin{array}{ll} & \text{IF } x_c \text{ THEN } s_1 \ (\mathit{option}\ x_b \ (\text{ELSE } s_2)) \\ \Longrightarrow & \text{CASE } x_c \text{ OF} \\ & \text{TRUE : } \ s_1 \\ & \mathit{option}\ x_b \ (\text{FALSE : } \ s_2). \end{array}$$

The condition of this transformation rule is $NEQ(s_2, \text{NULL})$. At first we transform the program segment IF f THEN $[a, b]$ ELSE $[c, d]$ using the rule by solving the following contextual matching problem:

$$\{(x_c, f), (s_1, [a, b]), (\mathit{option}\ x_b \ (\text{ELSE } s_2), \mathit{some}([c, d]))\} \ [NEQ(s_2, \text{NULL})].$$

This can be transformed to

$$\{(x_c, f), (s_1, [a, b]), (x_b, \mathit{true}), (s_2, [c, d])\} \ [NEQ([c, d], \text{NULL})].$$

The satisfiable match is

$$\{x_c \mapsto f, s_1 \mapsto [a, b], x_b \mapsto \mathit{true}, s_2 \mapsto [c, d]\}$$

and the transformation is

$$\text{IF } f \text{ THEN } [a, b] \text{ ELSE } [c, d] \ \Longrightarrow \ \begin{array}{l} \text{CASE } f \text{ OF} \\ \text{TRUE : } \ [a, b] \\ \text{FALSE : } \ [c, d]. \end{array}$$

Then we transform another program IF f THEN $[a, b]$ without "ELSE" part also using the rule by solving the contextual matching problem:

$$\{(x_f, f), (s_1, [a, b]), (\mathit{option}\ x_b \ (\text{ELSE } s_2), \mathit{none})\} \ [NEQ(s_2, \text{NULL})].$$

We get the following match:

$$\{(x_c, f), (s_1, [a, b]), (x_b, \mathit{false})\} \ [NEQ(s_2, \text{NULL})],$$

because there is no "ELSE" part in the program, so s_2 will not be substituted. The context $[NEQ(s_2, \text{NULL})]$ is suspended, but s_2 does not occur anywhere else in the program, so we can forget the context. The transformation is

$$\text{IF } f \text{ THEN } [a, b] \ \Longrightarrow \ \begin{array}{l} \text{CASE } f \text{ OF} \\ \text{TRUE : } \ [a, b]. \end{array}$$

3.4 Related Work

3.4.1 Contextual Rewriting

Contextual rewriting, as a more general case of the traditional conditional rewriting [50], has been studied by several papers. Even though the word "contextual rewriting" is not used in [10], the research about contextual rewriting is influenced by the simplification process and the cross-fertilization process of Boyer-Moore's theorem prover. In that paper, the idea of using the remaining literals in a clause as the context to simplify one literal was first described. In this chapter, contextual matching has been introduced to support program transformations by conditional rules.

Zhang and Kapur's contextual rewriting

We refer here to Zhang and Kapur's papers [88] and [87]. To simplify a term t by contextual rewriting, in addition to a set R of conditional rewriting rules, there is also a context C of t consisting of a set of equations. Under the assumption that the equations in C are true, C and R together rewrite t to another term. If C is empty, the contextual rewriting is identical to the traditional conditional rewriting. It is implemented in the theorem prover *RRL* – Rewrite Rule Laboratory – to support both deductive and inductive proving.

Given a set C of equations and a set R of rewrite rules, a **rewriting under the context** C is recursively defined as follows: A term t is rewritten to t' by contextual rewriting using R and C, denoted as $t \rightarrow_{C,R} t'$, if either:

- $t \approx_C t'$; or
- there exist a position p in t, a rewrite rule $l \rightarrow r \Leftarrow c$ in R and a substitution σ, such that $\sigma\ l = t|_p$, $t' = t[\sigma r]_p$ and $\sigma c \rightarrow^*_{C,R}\ true$,

where $\approx_C$ is a congruence relation: (i) $t \approx_C t'$ if $t = t' \in C$, (ii) $f \cdots t_1 \cdots \approx_C f \cdots t_2 \cdots$ if $t_1 \approx_C t_2$. From the definition we can see:

- Equations in contexts are used as additional 'rules' to simplify terms.
- Conditions in rewriting rules will also be proved through rewriting.

For example, R contains the rule:

$$gcd\ (x * y)\ x \rightarrow x.$$

Let C be the context $\{z = z_1 * z_2\}$ and $gcd\ z\ z_1$ be a term. Then

$$gcd\ z\ z_1\ \rightarrow_{C,R}\ gcd\ (z_1 * z_2)\ z_1\ \rightarrow_{C,R}\ z_1.$$

So technically, the contexts of matching problems and the contexts of terms to be rewritten are different: The contexts of matching problems are only used as constraints on matches, which will never be introduced into matching problems and will not result in additional narrowing steps. In contextual rewriting, terms in contexts can be transformed into terms to be rewritten, which may be rewritten with some rewrite rules later.

Remy's contextual rewriting

To the best of our knowledge, the name of contextual rewriting first appeared in Remy's thesis [72]. In Remy's work, a context is a Boolean expression.

Given a Boolean expression C and a set R of (conditional) rewrite rules, **rewriting** a term t to t' **under the context** C, denoted as $t \rightarrow_{C,R} t'$, if there exist a position p in t, a rule $l \rightarrow r \Leftarrow c$ in R, and a substitution σ such that

- $\sigma l = t|_p$,
- $(C \Rightarrow \sigma c) \rightarrow_R^* true$, where $\Rightarrow$ denotes the Boolean implication,
- $t' = t[\sigma r]_p$.

The definition shows:

- equations in contexts are used as additional conditions of rewriting;
- whether a context implies certain conditions of rewriting rules or not is proved again through rewriting.

Similarly, in contextual matching a context is also only used to establish conditions of its matches. One difference is that in our case, a context may have three states, "non-satisfiable", "satisfiable" and "suspended". In Remy's definition, context only have two states "true" for "satisfiable", and "false" for "non-satisfiable". In the last section we have seen that "suspended" contexts are useful for program transformations.

Ganzinger's contextual rewriting

Ganzinger's definition of contextual rewriting [24] pays particular attention to the fact that the variables in a context are treated as constants. To simplify a term t in the context C by a set R of rewriting rules, a skolem substitution θ is first applied to t and C, which replaces each variable of t and C by a new constant. Then θt is rewritten by the new rewriting system $R \cup (\theta C)$ using classical conditional rewriting. That is, $t \rightarrow_{C,R} t'$ iff $\theta t \rightarrow_{R \cup (\theta C)} \theta t'$.

This definition is simple and easy to understand, but C must also be a set of rewriting rules instead of a set of equations and disequations, and the properties of the rewriting system $R \cup (\theta C)$, such as confluence and termination, are also decided by C. Furthermore, to realize $\theta t \rightarrow_{R \cup (\theta C)} \theta t'$ is in general more complicated than to realize $t \rightarrow_R t'$.

To summarize, our approach has the following advantages:

- The method to solve context equations and disequations is simple and efficient.
- It is useful to consider suspended contexts, together with satisfiable and non-satisfiable contexts, in program transformations.
- The "prove during matching" method used in this chapter is more efficient than "matching then prove" (see [72, 88, 87]), particularly if $\mathcal{E}$-matching is considered.
- Any matching algorithm to solve usual matching problems can be used to solve matching problems with contexts without essential changes.

3.4.2 Conditional Narrowing and Unification

Conditional term rewriting systems are of growing importance in the field of term rewriting. For instance, a set of equational Horn clauses can be viewed as a conditional term rewriting system, where the rules are the clause heads and the conditions are the respective bodies. To solve the unification or matching problems associated with this kind of rewriting systems is thus an extension of the well-known results on the completeness of narrowing to the conditional case. The work done by Kaplan [45], Hussmann [42], Giovannetti and Moiso's [26] are some examples. In general, narrowing a term t at the position p with a conditional rule $l \rightarrow r \Leftarrow c$ consists of

- finding a most general unifier σ of $t|_p$ and l;
- proving the instance σc of the condition.

The main differences between different conditional narrowings are how and when the conditions in rewrite rules will be proved. Two frequently used methods are: (1) treating them as new unification problems, as in [45, 42]; (2) solving them during additional narrowing steps, as in [26].

In our present specifications, only unconditional equations are considered. So, in a narrowing step there is no additional condition to be proved. Suppose we have a conditional term rewriting system CR, for instance a set of program transformation rules, which uses function symbols defined by an unconditional specification R. The contextual matching algorithm solves contextual matching problems with respect to the unconditional term rewriting system R, where the contexts are the conditions of the conditional term rewriting system CR.

For instance, $l \rightarrow r \Leftarrow c$ is a conditional rule in CR, which contains function symbols defined in R. We want to rewrite a term t using the rule. Two things should be done:

- find a substitution such that $(\sigma l) \downarrow_R t$; i.e., solve the matching problem $\{(l,t)\}$; and
- prove the condition σc.

Combining these two things together, we have to solve the following contextual matching problem:

$$\{(l,t)\}\ [c].$$

Chapter 4

Extended Higher-Order Matching

In this chapter, we study the semantic matching problems associated with a class of higher-order specifications, which is an extension of the first-order specifications discussed in the previous chapters.

In recent years, there have been many successful attempts to lift results from first-order term rewriting to the higher-order case. Starting with the work of Klop [47], some of the results are: critical pairs of higher-order term rewriting systems [58]; confluence of orthogonal higher-order term rewriting systems [59]; termination criteria [69]; solving higher-order equations [70]; and the completeness of first-order narrowing strategies to higher-order patterns [71]. This interest in higher-order rewriting follows the progress in its applications, for instance, functional languages and theorem provers, where higher-order concepts are of growing attention. In this chapter, we lift another first-order concept — decidable matching for a class of higher-order specifications, and show its applications in program transformations.

Unification of typed λ-terms, also called higher-order unification, is the basic inference mechanism for many theorem provers and programming languages based on typed λ-calculus, such as [56, 67]. The implementations of most, if not all of these systems, rely on the unification algorithm invented by Huet [39]. Although the problem itself is undecidable and infinite sets of most general unifiers may exist, Huet's algorithm performs surprisingly well in practice. This observation lead Dale Miller to the discovery of a subclass of λ-terms, so called **(higher-order) patterns**, which behave almost like first-order terms with respect to unification: unification is decidable and unifiable terms have most general unifiers, which are easy to compute. In practice, these higher-order patterns occur very frequently, which explains why Huet's algorithm is well-behaved most of the time.

Recall that in our first-order matching algorithm, we use first-order syntactic unification during structural narrowing

$$t \leadsto_{l=r} t',$$

which means there exists a most general unifier σ of t and l such that $t' = \sigma r$. Now, if both t and l are higher-order patterns, then structural narrowing will work similarly.

Yet, higher-order patterns are often too restrictive for many applications. This holds particularly in equations. For instance, the following function *map* is a standard example for functional programs:

$$\begin{array}{lcl} map\ F\ nil & = & nil \\ map\ F\ (X :: X_s) & = & (F\ X) :: (map\ F\ X_s) \end{array}$$

Its second equation has a non-pattern right-hand side, see definition 4.3. Hence, rewriting or narrowing with this rule may yield non-pattern terms. In this chapter, we will first extend patterns to a class of **extended patterns** to settle this problem to a certain extent.

Of course, higher-order semantic matching problems are also dependent on higher-order matching problems. In the above example, $F\,X$ is a typical higher-order term. To ensure the decidability of the class of higher-order semantic matching problems, only decidable higher-order matching problems will be considered in this chapter, such as Huet and Lang's second-order matching [40].

4.1 Preliminaries

In this section, we present a number of basic definitions and results related to typed lambda calculus, which are the bases of higher-order specifications. Our notation is essentially consistent with [4], [36], and [23].

The simply typed λ calculus

In order to discuss higher-order signatures, we first have to introduce types. Although we already have types for first-order terms, types for higher-order terms are a little complicated. Assume that we have been given some symbols called **base types**, or sorts in the first-order case, then we define the set of **types** $\mathcal{T}$ as follows:

- each base type is a type;
- if τ_1 and τ_2 are types, then $\tau_1 \rightarrow \tau_2$ is a type.

In general, the base types will depend on the intended use of the system we wish to build. Each type $\tau_1 \rightarrow \tau_2$ denotes some given set of functions from τ_1 to τ_2. $\rightarrow$ is right-associative, so $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau$ denotes $\tau_1 \rightarrow (\cdots(\tau_n \rightarrow \tau)\cdots)$. The **order** of type τ is defined recursively by:

$$\begin{array}{rcll} \mathcal{ORD}(\tau) & = & 1, & \text{if } \tau \text{ is a base type} \\ & = & max_{1\leq i\leq n} Ord(\tau_i) + 1, & \text{if } \tau = \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_{n+1} \\ & & & \text{and } \tau_{n+1} \text{ is a base type.} \end{array}$$

A higher-order signature is similar to a first-order signature except that functions (but not constructors) may have higher-order types. Let $\mathcal{V} = \bigcup_{\tau\in\mathcal{T}} \mathcal{V}_\tau$ be a set of variables.

Definition 4.1 (Higher-order terms) For each type τ, the set of (higher-order) terms of type τ, denoted as $\mathcal{T}_\tau(\mathcal{V})$, is defined as follows:

- every variable in $\mathcal{V}_\tau$ is a term of type τ;
- if s and t are terms of types $\tau_1 \rightarrow \tau_2$ and τ_1 respectively, then $s\ t$ is a term of type τ_2;
- if x is a variable of type τ_1 and s is a term of type τ_2, then $\lambda x.s$ is a term of type $\tau_1 \rightarrow \tau_2$.

An occurrence of a variable x in a term t is **bound** iff it is in a part of t with the form $\lambda x.t$; otherwise it is **free**. If x has at least one free occurrence in t, it is called a **free variable** of t. The set of all free variables of t is denoted as $\mathcal{FV}(t)$. If $\mathcal{FV}(t) = \emptyset$, t is called a **ground term**.

The set $\mathcal{O}(t)$ of all occurrences of a term t is defined as in the first-order case.

$$
\begin{array}{lcll}
\mathcal{O}(\lambda x.t) & = & \{\Lambda\} \cup \{1.p \mid p \in \mathcal{O}(t)\} & \\
\mathcal{O}(a) & = & \{\Lambda\}, & \text{if } a \text{ is a variable or a constant} \\
\mathcal{O}(f\ t_1 \cdots t_n) & = & \{\Lambda\} \cup \{i.p | 1 \le i \le n \wedge p \in \mathcal{O}(t_i)\}, & \text{if } f \text{ is a constructor, a function} \\
 & & & \text{or a variable}
\end{array}
$$

If $p \in \mathcal{O}(t)$, then $\mathcal{BV}(t,p)$ denotes the set of all **bound variables** on the path from the root of t to the occurrence p. For example, let $t = \lambda x.\lambda y.(map\ (\lambda z.z + x)\ y)$, then

$$
\begin{array}{ll}
t|_{1.1} = map\ (\lambda z.z + x)\ y, & \mathcal{BV}(t, 1.1) = \{x, y\} \\
t|_{1.1.2} = y, & \mathcal{BV}(t, 1.1.2) = \{x, y\} \\
t|_{1.1.1.1} = z + x, & \mathcal{BV}(t, 1.1.1.1) = \{x, y, z\}
\end{array}
$$

A term t of type τ is said to be **n-order**, if the types of all its variables are no more than n, and the types of all its constants are no more than $n+1$. Consequently, a term t is said to be **first-order**, if each variable in t has a base type and each constant in t has a base or a second-order type. $map\ F\ X_s$ is a second-order term if the type of F is second-order; it is third-order if the type of F is third-order.

Convention:

- We write $((\cdots(a\ t_1)\cdots)\ t_n)$ as $a\ t_1 \cdots t_n$, and $\lambda x_1.\cdots.\lambda x_n.t$ as $\lambda x_1, \cdots, x_n.\ t$ simply.
- x, y and z are used for bound variables.
- Upper-case letters denote free variables, such as, X, Y, Z, F, and G.
- All terms are assumed to be simply typed. We write $t : \tau$ if t is of type τ.

Let $\sigma = \{x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\}$ be a substitution, and t a term, then σt is the result of substituting t_i for every free occurrence of x_i in t $(1 \le i \le n)$, and changing bound variables to avoid clashes. The precise definition is by induction on t and can be found in [13, 36]. The following are two examples:

$$
\begin{array}{lcll}
\{x \mapsto t, y \mapsto s\}(\lambda y.x\ y) & = & \lambda y.t\ y, & \text{if } y \notin \mathcal{FV}(t) \\
\{x \mapsto t, y \mapsto s\}(\lambda y.x\ y) & = & \lambda z.t\ z, & \text{if } y \in \mathcal{FV}(t) \wedge z \notin \mathcal{FV}(t)
\end{array}
$$

The **reduction relation** on λ-terms is usually defined by the following three rules:

$$
\begin{array}{lll}
(\alpha\text{-reduction}) & \lambda x.t \to_\alpha \lambda y.\{x \mapsto y\}t, & \text{if } y \notin \mathcal{FV}(t) \\
(\beta\text{-reduction}) & (\lambda x.t)\ s \to_\beta \{x \mapsto s\}t & \\
(\eta\text{-reduction}) & \lambda x.t\ x \to_\eta t, & \text{if } x \notin \mathcal{FV}(t)
\end{array}
$$

If a term t contains an occurrence of $(\lambda x.t)s$ and we replace that occurrence by $\{x \mapsto s\}t$, and the result is t', we say t β-**reduces to** t' or $t \to_\beta t'$, and $(\lambda x.t)s$ is a β-**redex**. $t \to_\eta t'$ and $t \to_{\beta\eta} t'$ are defined similarly, where $\beta\eta$ means β or η. Let γ be one of α, β, η, and $\beta\eta$. A term t that contains no γ-redices is called a γ-**normal form**. We also define the reflexive and transitive closure $\rightarrow_\gamma^*$ in the obvious way. If $t \leftrightarrow_\gamma^* t'$, then t and t' are γ-**equivalent**. There are two important properties of the $\beta\eta$-reduction: confluence and strong-normalization.

Theorem 4.1 (Church-Rosser theorem) If $t \rightarrow_{\beta\eta}^* t_1$ and $t \rightarrow_{\beta\eta}^* t_2$, then there exists t' such that

$$t_1 \rightarrow_{\beta\eta}^* t' \text{ and } t_2 \rightarrow_{\beta\eta}^* t'.$$

□

Theorem 4.2 (Strongly normalizing) In the system of typed λ-terms, there is no infinite sequence of $\beta\eta$-reductions. □

As a consequence, each term t has an (up to α-equivalence) unique β-normal form, denoted by $t\downarrow_\beta$. In particular, each term in β-normal form can be presented as

$$\lambda x_1, \cdots, x_n.(a\ t_1 \cdots t_m),$$

where the **head** a is an **atom**, i.e., a constructor, a function, or a variable, and the terms $t_1, \cdots, t_m$ are all in the same form. $\lambda x_1, \cdots, x_n$ is called the λ-**binder** of t.

Definition 4.2 (Rigid and flexible terms) A term whose head is a function, a constructor, or a bound variable is called a rigid term; if its head is a free variable it is a flexible term.

For example, the term $\lambda x.F(\lambda y.(y\ x))$ is flexible, because F is a free variable, but its subterm $\lambda y.(y\ x)$ is rigid.

To "factor out" η-reduction by considering only η-equivalence classes of terms, we define a class of η-expanded form. Let $t = \lambda x_1, \cdots, x_n.(a\ t_1 \cdots t_m)$ be a term in β-normal form of type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_{n+1} \rightarrow \cdots \rightarrow \tau_{n+k} \rightarrow \tau$ with τ a base type. Then the η-**expanded form** of t, denoted by $t\uparrow$, is

$$\lambda x_1, \cdots, x_n, x_{n+1}, \cdots, x_{n+k}.\ (a\ t_1\!\uparrow \cdots\ t_m\!\uparrow\ x_{n+1}\!\uparrow \cdots\ x_{n+k}\!\uparrow)$$

where $x_{n+1} : \tau_{n+1}, \cdots, x_{n+k} : \tau_{n+k}$ are all new variables. For instance,

$$map\!\uparrow = \lambda x, y.(map\ (\lambda z.x\ z)\ y)$$

where x: $\tau_1 \rightarrow \tau_2$, y: $list_1$, z: τ_1, if map: $(\tau_1 \rightarrow \tau_2) \rightarrow list_1 \rightarrow list_2$.

The η-expanded form of a term is more useful than its η-normal form because it makes the type of the term and all its subterms more explicit, and is therefore a convenient syntactic convention for representing the congruence class of all terms modulo the η-equivalence. It is easy to show that for any two terms s and t in β-normal form, we have

$$s \leftrightarrow_\eta^* t \ \text{ iff } \ s\!\uparrow \leftrightarrow_\alpha^* t\!\uparrow .$$

The η-expanded form and β-normal form of any term t is called a **long $\beta\eta$-normal form**, denoted as $t\updownarrow$.

Theorem 4.3 For every two terms s and t, $s \leftrightarrow_{\beta\eta}^* t$ iff $s\updownarrow \leftrightarrow_\alpha^* t\updownarrow$. □

Convention:

- We do not distinguish α-equivalent terms.
- We shall assume that terms under discussion are in long $\beta\eta$-normal form unless otherwise stated.
- In general, substitutions are assumed to be normalized in this thesis. So if we say $\sigma = \theta$, it means $(\sigma x)\updownarrow \leftrightarrow_{\alpha}^{*} (\theta x)\updownarrow$.

Higher-order patterns

Higher-order patterns are an important subclass of λ-terms, which include bound variables, but behave in many aspects almost as first-order terms.

Definition 4.3 (Higher-order patterns) A term t in β-normal form is a (higher-order) pattern if every free occurrence of a variable F is in a subterm $F\ t_1 \cdots t_n$ of t such that the t_i $(1 \le i \le n)$ are η-equivalent to a list of distinct bound variables.

Examples of higher-order patterns are

$$\lambda x.c\ x, \quad F, \quad \lambda x.F(\lambda z.x\ z), \quad \lambda x, y.F\ y\ x.$$

Examples of non-patterns are

$$F\ c, \quad \lambda x.F\ x\ x, \quad \lambda x.F(F\ x);$$

in the first term, c is a constant; in the second term, the two arguments of F are the same bound variable; in the third term, $F\ x$ is not η-equivalent to a bound variable. The following crucial result about unification of patterns is due to Miller [55].

Theorem 4.4 It is decidable whether two patterns are unifiable or not; if they are unifiable, a most general unifier can be computed. □

A unification algorithm for patterns can be found in appendix A.

Higher-order term rewriting systems

Higher-order term rewriting systems are generalizations of first-order rewriting systems [16, 47] to terms with higher-order functions and bound variables. There are several different definitions of higher-order term rewriting systems. We will adopt the definition from Nipkow, see [58, 59]. The reasons to use this definition are:

- It restricts Klop's combinatory reduction systems by considering only λ-calculus.
- The rewriting relations are quite simple.
- It is appropriate for our discussion of extended matching problems.

The other two definitions from Klop [46] and Wolfram [84] are presented in section 4.6.

Definition 4.4 (Higher-order rewriting systems) A rewrite rule $l \to r$ is higher-order, if

- l is a pattern, but not η-equivalent to a free variable;
- l and r are of the same base type, and $\mathcal{FV}(r) \subseteq \mathcal{FV}(l)$.

A higher-order term rewriting system $\langle \Sigma, \mathcal{R} \rangle$ consists of a higher-order signature Σ and a finite set $\mathcal{R}$ of higher-order rewrite rules.

According to this definition, the left-hand side of each rule has the form $f\ l_1\ \cdots\ l_n$, where f is a function, a variable, or a constructor.

A higher-order rewriting system R induces a rewriting relation $\to_{\mathcal{R}}$ on terms in long $\beta\eta$-normal form:

$$s \to_{\mathcal{R}} t \text{ iff } \exists (l \to r) \in \mathcal{R}, p \in \mathcal{O}(s), \sigma.\ s|_p = (\sigma l)\updownarrow\ \wedge\ t = s[(\sigma r)\updownarrow]_p.$$

The following is an example:

$$\begin{array}{rcl} \lambda x.(map\ (\lambda y.succ\ y)\ (x :: nil)) & \to & \lambda x.(succ\ x) :: (map\ (\lambda y.succ\ y)\ nil) \\ & \to & \lambda x.(succ\ x) :: nil. \end{array}$$

The first rule used is $map\ F\ (X :: X_s) \to (F\ X) :: (map\ F\ X_s)$, the position is 1, and the substitution is

$$\{F \mapsto \lambda y.succ\ y, X \mapsto x, X_s \mapsto nil\}.$$

The second rule used is $map\ F\ nil \to nil$.

To narrow terms with higher-order rules, we usually need to rename free variables. Different from the renaming of first-order rules, we should rename free variables and lift their types as well. A $\{x_1, \cdots, x_k\}$-**lifter** of a term t **away from** W is a substitution

$$\sigma = \{F \mapsto (\rho F)\ x_1 \cdots x_k \mid F \in \mathcal{FV}(t)\},$$

where

- ρ is a renaming such that $\mathcal{D}(\rho) = \mathcal{FV}(t)$,
- $\mathcal{I}(\rho) \cap W = \emptyset$,
- $\rho F : \tau_1 \to \cdots \to \tau_k \to \tau$ if $x_i : \tau_i$ $(1 \leq i \leq k)$, and $F : \tau$.

The $\{x_1, x_2\}$-lifter of the term $map\ F\ (X :: X_s)$ away from $\emptyset$ is

$$\{F \mapsto F'\ x_1\ x_2, X \mapsto X'\ x_1\ x_2, X_s \mapsto X'_s\ x_1\ x_2\},$$

where F', X', and X'_s are free variables with proper types.

Let $l_1 \to r_1$ and $l_2 \to r_2$ be two rules in a rewriting system R; they are **overlapping** if there exists a position $p \in \mathcal{O}(l_1)$ such that

- the head of $l_1|_p$ is not variable free in l_1, and
- $\lambda x_1, \cdots, x_k.\ l_1|_p$ and $\lambda x_1, \cdots, x_k.\ \theta l_2$ are unifiable, where $\{x_1, \cdots, x_k\} = \mathcal{BV}(l_1, p)$ and θ is a $\{x_1, \cdots, x_k\}$-lifter of l_2 away from $\mathcal{FV}(l_1)$.

A higher-order rewriting system $\langle \Sigma, \mathcal{R} \rangle$ is **orthogonal** if it is left-linear and any two different rules in $\mathcal{R}$ are non-overlapping. The following theorem is due to Nipkow [59].

Theorem 4.5 If R is an orthogonal higher-order term rewriting system, then $\rightarrow_{\mathcal{R}}$ is confluent. $\square$

Convention:

- If we say $\langle \Sigma, \mathcal{E} \rangle$ is a **higher-order specification**, we mean it is a higher-order rewriting system by treating equations as rewrite rules from left to right.
- In this chapter, we will sometimes omit "higher-order" and simply say "rewriting system", "specification", and so on.

4.2 Higher-Order Constructor-Based Specifications

The concepts defined for first-order specifications will be lifted to the higher-order case in this section. At first, we give an example of higher-order constructor-based specifications.

$$
\begin{array}{rl}
\mathit{Map}[\alpha_1, \alpha_2] \quad = & \mathit{List}[\alpha_1]\{\mathit{list}\ \textbf{with}\ \mathit{list}_1\}\ + \\
& \mathit{List}[\alpha_2]\{\mathit{list}\ \textbf{with}\ \mathit{list}_2\}\ + \\
\textbf{funs}: & \mathit{map}\ :\ (\alpha_1 \to \alpha_2) \to \mathit{list}_1 \to \mathit{list}_2 \\
\textbf{eqns}: & \forall\ X \in \alpha_1;\ X_s \in \mathit{list}_1;\ F \in \alpha_1 \to \alpha_2 : \\
& \mathit{map}\ F\ \mathit{nil} = \mathit{nil} \\
& \mathit{map}\ F\ (X :: X_s) = (F\ X) :: (\mathit{map}\ F\ X_s)
\end{array}
$$

$\mathit{Map}[\alpha_1, \alpha_2]$ is an orthogonal higher-order rewriting system if we treat equations as rewrite rules.

Extended patterns

We should first extend the concept of patterns, since the definition of patterns are often too restrictive to define functions. For example, $(F\ X) :: (\mathit{map}\ F\ X_s)$ is not a pattern. Extended patterns are a subclass of λ-terms, which are patterns or can be transformed to patterns easily.

A term t is called a **pure λ-term** (short: **pure term**) if t contains no function symbols. $F\ X$, $\lambda x.(G\ x\ x)$, and $\lambda x.F(F\ x)$ are examples of pure terms, note that they are not patterns. $\lambda x.F\ x$ is a pure term and a pattern as well. $\mathit{map}\ F\ X_s$ and $\lambda x, y.x + y$ are not pure terms, since $+$ and map are function symbols.

Definition 4.5 (Extended patterns) A term t is an extended pattern, if there exists a list of pair-wise disjoint positions $p_1, \cdots, p_k \in \mathcal{O}(t)$ such that

- each $t|_{p_i}$ $(1 \le i \le k)$ is a pure term but not a pattern, and has the form $F\ s_1 \cdots s_m$ with F a free variable;
- $t[s_1, \cdots, s_k]_{\{p_1, \cdots, p_k\}}$ is a pattern, where each s_i $(1 \le i \le k)$ has the form $F_i\ y_{i,1} \cdots y_{i,n_i}$ such that
  - F_i is a new free variable,
  - $\{y_{i,1}, \cdots, y_{i,n_i}\} = \mathcal{BV}(t, p_i)$.

The **pattern abstraction** of t is $t|_{\{p_1, \cdots, p_n\}}$.

This definition shows that if some pure subterms of an extended pattern are replaced, it will become a pattern. For instance, the pattern abstraction of $(F\ X) :: (\mathit{map}\ F\ X_s)$ is $(F\ X) :: (\mathit{map}\ F\ X_s)|_{\{1\}}$, $Y :: (\mathit{map}\ F\ X_s)$ is a pattern; and the pattern abstraction of

$$t = \lambda xs.((F\ X) :: ((F\ Y) :: (\mathit{map}\ F\ xs)))$$

is $t|_{\{1.1,\ 1.2.1\}}$, its associated pattern is

$$\lambda xs.((G_1\ xs) :: ((G_2\ xs) :: (\mathit{map}\ F\ xs))).$$

Lemma 4.1 If t is an extended pattern, then its pattern abstraction is unique.

Proof. Let $t|_{\{p_1,\cdots,p_{k_1}\}}$ and $t|_{\{q_1,\cdots,q_{k_2}\}}$ be two pattern abstractions of t. Then each t_{p_i} $(1 \leq i \leq k_1)$ is not a pattern and has the form $F\ s_1 \cdots s_n$, where F is a free variable. Because $t|_{\{q_1,\cdots,q_{k_2}\}}$ is also a pattern abstraction, so there must exist a $1 \leq j \leq k_2$ such that q_j is a prefix of p_i. Because $t|_{q_j}$ has the form $G\ t_1 \cdots t_m$, where G is a free variable, so q_j must equal to p_i, otherwise $t|_{\{p_1,\cdots,p_{k_1}\}}$ is not a pattern abstraction. That is, $t|_{\{p_1,\cdots,p_{k_1}\}}$ and $t|_{\{q_1,\cdots,q_{k_2}\}}$ are the same pattern abstraction of t. □

Definition 4.6 (Higher-order constructor-based specifications) Let $\mathcal{HSP} = \langle \Sigma, \mathcal{E} \rangle$ be an orthogonal higher-order specification. $\mathcal{HSP}$ is said to be a constructor-based specification, if

- for each equation $l = r$ in $\mathcal{E}$, l has the form $f\ l_1 \cdots l_n$, where f is a function symbol in Σ, and each l_i $(1 \leq i \leq n)$ contains no function symbols.

Let $\mathcal{HSP}$ be a constructor-based specification; a function $f \in \Sigma$ is a **(higher-order) matching combinator**, if

- r is an extended pattern, for each $l = r \in \mathcal{E}_f$.

The requirement that r is an extended pattern makes the separation of pure higher-order subterms from r possible, which transforms r into a pattern. *Map* is a constructor-based specification, and *map* is a matching combinator. In the following constructor-based specification *Fold*, *foldl* is a matching combinator, but *foldr* is not, since $F\ Y\ (\mathit{foldr}\ F\ Z\ Y_s)$ is not an extended pattern.

$$
\begin{array}{rl}
\mathit{Fold}[\alpha_1, \alpha_2] \ = & \mathit{List}[\alpha_1] + \\
\textbf{funs}: & \mathit{foldl}, \mathit{foldr}\ :\ (\alpha_1 \to \alpha_2 \to \alpha_2) \to \alpha_2 \to \mathit{list} \to \alpha_2 \\
\textbf{eqns}: & \forall\ F \in \alpha_1 \to \alpha_2 \to \alpha_2;\ Y \in \alpha_1;\ Z \in \alpha_2;\ Y_s \in \mathit{list}: \\
 & \mathit{foldl}\ F\ Z\ \mathit{nil} = Z \\
 & \mathit{foldl}\ F\ Z\ (Y :: Y_s) = \mathit{foldl}\ F\ (F\ Z\ Y)\ Y_s \\
 & \\
 & \mathit{foldr}\ F\ Z\ \mathit{nil} = Z \\
 & \mathit{foldr}\ F\ Z\ (Y :: Y_s) = F\ Y\ (\mathit{foldr}\ F\ Z\ Y_s)
\end{array}
$$

From these examples, we can see that the class of higher-order matching combinators contains many practically interesting functions.

Proper non-decreasing functions

To define a class of higher-order matching combinators which have finite semantic matching problems, we will introduce so called proper non-decreasing functions.

A **suitable property** P, as in the first-order case, is a measure for terms in long $\beta\eta$-normal form along with a well-founded partial ordering $\rhd$, which compares values of P. It has the following properties:

- $P(X) \not\rhd P(a)$ for any variable X and constant a.
- $P(\lambda x.t) \rhd P(\lambda x.s)$ if $P(t) \rhd P(s)$.

- For any term s, $p \in \mathcal{O}(s)$, and $\mathcal{BV}(s,p) = \{x_1, \cdots, x_k\}$,

$$P(s) \ \rhd \ P(\lambda x_1, \cdots, x_k.s|_p)$$

  if $\lambda x_1, \cdots, x_k.s|_p \neq s$.

The **depth** of terms can also to be extended to long $\beta\eta$-normal forms by the following definition:

$$\begin{array}{lcll} depth(\lambda x.s) & = & depth(s) & \\ depth(a) & = & 0, & \text{if } a \text{ is a variable or a constant} \\ depth(f\ t_1 \cdots\ t_n) & = & \max_{1 \leq i \leq n} depth(t_i) + 1 & \text{if } f \text{ is a function, a constructor,} \\ & & & \text{or a free variable} \end{array}$$

It is not difficult to see that the *depth* is still a suitable property, along with the greater-then relation ($>$) on natural numbers. For instance, $\lambda x.F(\lambda y.y\ x)|_{1.1} = \lambda y.y\ x$, we also have $depth(\lambda x.F(\lambda y.y\ x)) > depth(\lambda x, y.y\ x)$.

Definition 4.7 (Proper non-decreasing functions) Let P be a suitable property along with the well-founded partial ordering $\rhd$. $f\ t_1 \cdots t_n$ is a pattern, where f is a function symbol, each t_i $(1 \leq i \leq n)$ is a normal form containing no function symbols, and t is any normal form. f is said to be proper non-decreasing (with respect to P) if whenever σ is a minimal match of $f\ t_1 \cdots t_n$ and t, then $P(\sigma X) \not\rhd P(t)$, for all $X \in \mathcal{D}(\sigma)$. Any function which does not have this property is called a **proper decreasing** function (with respect to P).

We have defined that an equation $l = r$ is variable-dropping, if $\mathcal{FV}(r) \subset \mathcal{FV}(l)$. For instance, $map\ F\ nil = nil$ from the specification *Map* is a variable-dropping equation.

In higher-order specifications, there is another kind of variable-dropping equations. For example, the equation $map\ F\ (X :: X_s) = (F\ X) :: (map\ F\ X_s)$ presents a set of equations for each concrete F. Let $F = \lambda y.t$ such that $y \notin \mathcal{FV}(t)$, then it is the equation:

$$map\ (\lambda y.t)\ (X :: X_s) \ = \ t :: (map\ (\lambda y.t)\ X_s).$$

X does not occur on the right-hand side of the equation, which is caused by the β-reduction $(\lambda y.t)\ X \rightarrow_\beta t$. For functions containing such kind of variable-dropping equations, the above definition is still valid. For instance, we can show that *map* is proper non-decreasing. Let $\sigma = \{X_1 \mapsto \lambda x.0,\ X_2 \mapsto 0 :: ((succ\ 0) :: nil)\}$,

$$\sigma(map\ X_1\ X_2) =_{\mathcal{E}_{map}} 0 :: (0 :: nil),$$

but $depth(0 :: ((succ\ 0) :: nil)) > depth(0 :: (0 :: nil))$, because σ is not a minimal match of $map\ X_1\ X_2$ and $0 :: (0 :: nil)$. Let $\sigma' = \{X_1 \mapsto \lambda x.0,\ X_2 \mapsto Y_1 :: (Y_2 :: nil)\}$, obviously $\sigma' \leq \sigma$,

$$\sigma'(map\ X_1\ X_2) =_{\mathcal{E}_{map}} 0 :: (0 :: nil),$$

and $depth(Y_1 :: (Y_2 :: nil)) \not> depth(0 :: (0 :: nil))$ by the assumption that $depth(Y_1)$, $depth(Y_2) \not> depth(0)$.

Formally, let σ be a minimal match of $map\ t_1\ t_2$ and b, where b is an $\mathcal{E}$-normal form.

- If b is *nil*, then $t_2 = nil$ or $t_2 = X_s$ and $\sigma X_s = nil$. For any other $Y \neq X_s \in \mathcal{D}(\sigma)$, $\sigma Y = Y$, otherwise σ is not a minimal match. So, $depth(\sigma Y) \not> depth(b)$.

- If b is $b' :: bs$, then $t_2 = t'_2 :: ts$ or $t_2 = X_s$ and $\sigma X_s = t'_2 :: ts$. $\sigma(t_1\ t'_2) = b'$ and $\sigma(map\ t_1\ ts) = bs$.
  - If $\sigma t_1 = \lambda x.t'_1$ and $x \notin \mathcal{FV}(t'_1)$, then $\sigma(t_1\ t'_2) = \sigma t'_1 = b'$. $depth(\sigma Y) \not> depth(b')$ for any $Y \in \mathcal{FV}(t'_1)$, since t_1 is a pattern. For any $Y \in \mathcal{FV}(t'_2)$, $\sigma Y = Y$ (σ is a minimal match).
  - If $\sigma t_1 = \lambda x.t'_1$ and $x \in \mathcal{FV}(t'_1)$, then $\sigma(t_1\ t'_2) = \sigma(\{x \mapsto t'_2\}t'_1) = b'$. Because both t_1 and t_2 are patterns, so $depth(\sigma Y) \not> depth(b')$ for any $Y \in \mathcal{FV}(t_1) \cup \mathcal{FV}(t_2)$. □

Higher-order acyclic matching combinators

With all these preparations, we can now introduce the following definition.

Definition 4.8 (Higher-order acyclic matching combinators) Let P be some suitable property, $\langle \Sigma, \mathcal{E} \rangle$ a constructor-based specification, f a matching combinator, and $l = r \in \mathcal{E}_f$. We call f a (higher-order) acyclic matching combinator, if

- the head of r is a variable or a constructor;
- any other function in r is an acyclic matching combinator; and
- there is no defined function in r nested below any function that is proper decreasing with respect to P.

This definition is very similar to the first-order case. In the above examples, *map* and *foldl* are both higher-order acyclic matching combinators. The following specification is a generalization of *Nat*,

$$
\begin{array}{lll}
GNat & = & \\
 & \textbf{sorts}: & gnat \\
 & \textbf{cons}: & c_1 \ :\ gnat \\
 & & c_2 \ :\ gnat \rightarrow gnat \\
 & \textbf{funs}: & gadd,\ gmul \ :\ (gnat \rightarrow gnat) \rightarrow gnat \rightarrow gnat \rightarrow gnat \\
 & \textbf{eqns}: & \forall\ F \in gnat \rightarrow gnat;\ X, Y \in gnat: \\
 & & gadd\ F\ c_1\ Y = F\ Y \\
 & & gadd\ F\ (c_2\ X)\ Y = c_2(gadd\ F\ X\ Y) \\
 & & \\
 & & gmul\ F\ c_1\ Y = F\ c_1 \\
 & & gmul\ F\ (c_2\ X)\ c_1 = F\ c_1 \\
 & & gmul\ F\ (c_2\ X)\ (c_2\ Y) = c_2(gadd\ F\ X\ (gmul\ F\ (c_2\ X)\ Y))
\end{array}
$$

In fact, it presents a set of specifications. If we take c_1 and c_2 as the constructors 0 and *succ* of natural numbers, and F as identity, then *gadd* and *gmul* are the addition $+$ and the multiplication $*$ on natural numbers respectively. Let $X = c_2(c_2(\cdots(c_2\ c_1)\cdots))$, then

$$gadd\ F\ X\ Y \ =_{\mathcal{E}}\ c_2(c_2(\cdots(c_2\ (F\ Y))\cdots)).$$

Using this fact, it is not difficult to prove that *gadd* is a *depth* proper non-decreasing function. So both *gadd* and *gmul* are acyclic matching combinators.

We emphasize here that the class of extended patterns enlarges patterns with terms which can be transformed to patterns easily. Furthermore, the requirement that the right-hand side of each equation should be an extended pattern prohibits equations of the form

$$f\ l_1 \cdots l_n \ = \ F\ \cdots (f\ t_1 \cdots t_n) \cdots$$

This kind of equations may cause infinite narrowing derivations, since the higher-order free variable F may be substituted by functions like $\lambda \cdots x_i \cdots .\ x_i$.

4.3 The Matching Algorithm $\mathcal{MA}^2$

Before presenting the algorithm, we make an important assumption: For the class of pure λ-terms concerned, there exists a matching algorithm that always produces a finite complete set of minimal matches for any matching problem. For instance, Huet and Lang's matching algorithm [40] always produces a finite complete set of minimal matches for second-order matching problems.

4.3.1 Higher-Order Structural Narrowing

To present the higher-order matching algorithm, we should first lift the first-order structural narrowing to a higher-order one.

Definition 4.9 (Higher-order structural narrowing) Let t be a pattern $\lambda x_1, \cdots, x_k.s$, where the head of s is not a free variable. t is (higher-order) structurally narrowable into t' with the equation $l = r$ and the substitution σ, if

- θ is a $\{x_1, \cdots, x_k\}$-lifter of l away from $\mathcal{FV}(t)$;
- σ is the most general unifier of t and $\lambda x_1, \cdots, x_k.\theta l$; and
- $t' = \sigma(\lambda x_1, \cdots, x_k.\theta r)$.

Because l is a pattern, so is $\lambda x_1, \cdots, x_k.\theta l$, since θ has the form

$$\{X_1 \mapsto F_1\ x_1 \cdots x_k, \cdots, X_m \mapsto F_m\ x_1 \cdots x_k\}$$

if $X_1, \cdots, X_m$ are free variables in l, where $F_1, \cdots, F_m$ are free variables not occurring in t. That is, the most general unifier of $\lambda x_1, \cdots, x_k.s$ and $\lambda x_1, \cdots, x_k.\theta l$ exists if they are unifiable. We still use $\leadsto$ to denote the higher-order structural narrowing relation. The following is an example of structural narrowing:

$$\begin{aligned} & t = \lambda x_1, x_2.(map\ G\ (x_1 :: (x_2 :: nil))) \\ & \leadsto_{map\ F\ (X::X_s)=(F\ X)::(map\ F\ X_s)}\ \lambda x_1, x_2.((G\ x_1) :: (map\ G\ (x_2 :: nil))) = t' \end{aligned}$$

where

- the $\{x_1, x_2\}$-lifter of $map\ F\ (X :: X_s)$ away from $\mathcal{FV}(t)$ is

  $$\theta = \{F \mapsto F'\ x_1\ x_2, X \mapsto X'\ x_1\ x_2, X_s \mapsto X_s'\ x_1\ x_2\}$$

  where F', X' and X_s' are free variables;
- the types of F', X' and X_s' are

  $$\begin{aligned} F' &: \ \alpha_1 \to \alpha_1 \to \alpha_1 \to \alpha_2 \\ X' &: \ \alpha_1 \to \alpha_1 \to \alpha_1 \\ X_s' &: \ \alpha_1 \to \alpha_1 \to list_1 \end{aligned}$$

  if map has the type:

  $$(\alpha_1 \to \alpha_2) \to list_1 \to list_2;$$

- the most general unifier of t and $\lambda x_1, x_2.\theta(map\ F\ (X :: X_s))$ is

$$\sigma = \{F' \mapsto \lambda x_1, x_2.G, X' \mapsto \lambda x_1, x_2.x_1, X'_s \mapsto \lambda x_1, x_2.(x_2 :: nil)\};$$

- $t' = \sigma(\lambda x_1, x_2.\theta((F\ X) :: (map\ F\ X_s)))$.

As in the first-order case, we will abstract patterns to terms that contain only one occurrence of a matching combinator to make structural narrowing work.

Definition 4.10 (Combinator abstractions) A combinator abstraction of a pattern t is defined as $t|_{\{p_1,\cdots,p_k\}}$ such that:

- $p_1, \cdots, p_k \in \mathcal{O}(t)$, and they are pairwise disjoint;
- each $t|_{p_i}$ has the form $f\ s_1 \cdots s_n$ with f a matching combinator; and
- $t[s_1, \cdots, s_k]_{\{p_1,\cdots,p_k\}}$ contains only one occurrence of a matching combinator, denoted as [t], where each s_i $(1 \leq i \leq k)$ has the form $F_i\ y_{i,1} \cdots y_{i,n_i}$ such that
  - F_i is a new free variable,
  - $\{y_{i,1}, \cdots, y_{i,n_i}\} = \mathcal{BV}(t, p_i)$.

Let

$$t = \lambda xs, ys.((map\ F\ xs) +\!\!+ (map\ G\ ys)).$$

The combinator abstraction of t is $t|_{\{1.1.1,\ 1.1.2\}}$, and

$$[t] = \lambda xs, ys.((F_1\ xs\ ys) +\!\!+ (G_1\ xs\ ys)).$$

For extended patterns, there are two possible abstractions: pattern abstraction that transforms an extended pattern to a pattern, and combinator abstraction that abstracts a pattern to a term containing only one occurrence of a matching combinator. In the sequel, if we write $\lambda x_1, \cdots, x_k.(f\ s_1 \cdots s_n)$, where f is a matching combinator, we mean that it is a pattern and each s_i contains no matching combinators. Otherwise, it will be written as its pattern abstraction or its combinator abstraction.

4.3.2 Matching by Transformations on System $\mathcal{TS}^2$

The matching algorithm $\mathcal{MA}^2$ is based on the following transformation system $\mathcal{TS}^2$ consisting of six rules. It starts with a ground matching problem $\Gamma = \{(s_1, t_1), \cdots, (s_n, t_n)\}$, where each s_i $(1 \leq i \leq n)$ is an extended pattern. It terminates with a solved matching problem Γ', if Γ is solvable. We make the following assumptions:

- Any matching pair is automatically α-converted such that both sides always have the same sequence of outermost λ-binders.
- Any term is always long $\beta\eta$-normalized.
- Any term is always $\mathcal{E}$-normalized.

We still use Γ_σ to denote the solved matching problem of the substitution σ, and σ_Γ to denote the substitution formed from the solved matching problem Γ.

(**Remove**) removes a trivial matching pair:

$$\{(t,t)\} \cup \Gamma \implies \Gamma.$$

(**Deletion**) deletes a matching pair playing no roles in solutions:

$$\{(\lambda x_1, \cdots, x_k.t, \lambda x_1, \cdots, x_k.(F\ x_1 \cdots x_k))\} \cup \Gamma \Longrightarrow \Gamma,$$

if $F \notin \mathcal{FV}(\Gamma)$.

(**Decomposition**) breaks a matching pair down into simpler ones:

$$\begin{array}{l}\{(\lambda x_1, \cdots, x_k.(c\ s_1 \cdots s_n), \lambda x_1, \cdots, x_k.(c\ t_1 \cdots t_n))\} \cup \Gamma \\ \Longrightarrow \{(\lambda x_1, \cdots, x_k.s_1, \lambda x_1, \cdots, x_k.t_1), \cdots, (\lambda x_1, \cdots, x_k.s_n, \lambda x_1, \cdots, x_k.t_n)\} \cup \Gamma,\end{array}$$

where $c \in \mathcal{C} \cup \mathcal{F} \cup \{x_1, \cdots, x_k\}$.

(**Abstraction**) abstracts a matching pair:

$$\{(s|_{\{p_1,\cdots,p_k\}}, t)\} \cup \Gamma \Longrightarrow \{(s', t), (s_1, t_1), \cdots, (s_k, t_k)\} \cup \Gamma,$$

where $s|_{\{p_1,\cdots,p_n\}}$ is a pattern abstraction or a combinator abstraction. Let $\mathcal{BV}(s, p_i) = \{y_{i,1}, \cdots, y_{i,n_i}\}$, and F_i be a new free variable, for each $1 \le i \le k$, then

$$\begin{array}{rcl} s_i &=& \lambda y_{i,1}, \cdots, y_{i,n_i}.s|_{p_i}, \\ t_i &=& \lambda y_{i,1}, \cdots, y_{i,n_i}.(F_i\ y_{i,1} \cdots y_{i,n_i}), \\ s' &=& s[F_1\ y_{1,1} \cdots y_{1,n_1}, \cdots, F_k\ y_{k,1} \cdots y_{k,n_k}]_{\{p_1,\cdots,p_k\}}. \end{array}$$

(**Matching**) solves a matching pair:

$$\{(s,t)\} \cup \Gamma \implies \Gamma_\sigma \cup (\sigma\Gamma),$$

if s is a pure λ-term, where σ is a minimal match of s and t.

(**Narrowing**) realizes structural narrowing of a matching pair:

$$\{(\lambda x_1, \cdots, x_k.(f\ s_1 \cdots s_n), t)\} \cup \Gamma \Longrightarrow \Gamma_\sigma\ \cup\ \{(s,t)\}\ \cup\ (\sigma\Gamma),$$

if f is a matching combinator, and $\lambda x_1, \cdots, x_k.(f\ s_1 \cdots s_n)$ is structurally narrowable to s with an equation $l = r \in \mathcal{E}_f$ and with the substitution σ.

In the following we give some explanations, since some of our transformation rules do not occur anywhere else, and some of the rules are different from the corresponding rules in the system $\mathcal{TS}^1$.

- Rule (Abstraction) is twofold. On the one hand, it separates a matching pair into a pattern part and several pure higher-order parts — pattern abstraction. On the other hand, it transforms a matching pair into one which contains only one occurrence of a matching combinator — combinator abstraction. Both of them are preparations for structural narrowing. According to its definition, rule (Abstraction) can only be applied to any matching pair at most two times. One is pattern abstraction, and the other is combinator abstraction. In $\mathcal{TS}^1$ we only have combinator abstraction.

- Rule (Matching) is new, which substitutes rule (Variable) in $\mathcal{TS}^1$. According to the assumption, the complete set of minimal matches of s and t exists if s is a pure λ-term.
- In rule (Narrowing), the term t to be narrowed is a pattern. So, if it is structurally narrowable with an equation in $\mathcal{E}_f$, then the most general unifier σ exists. Furthermore, for any X, σX contains no matching combinators, since the only occurrence of a function symbol on the left-hand side of each rule is at its head position.
- The rules (Remove), (Deletion), and (Decomposition) are the higher-order correspondences of (Remove), (Deletion), and (Decomposition) in $\mathcal{TS}^1$ respectively.

We give an example. Let

$$\begin{array}{rcl} t & = & \lambda x, y.\ (map\ G\ (x :: (y :: nil))) \\ s & = & \lambda x, y.\ ((succ\ x) :: ((succ\ y) :: nil)) \end{array}$$

be two terms to be matched. To narrow t with the equation

$$map\ F\ (X :: X_s)\ =\ (F\ X) :: (map\ F\ X_s),$$

at first, we $\{x, y\}$-lift the left-hand side of the equation to

$$l'\ =\ \lambda x, y.\ (map\ (F'\ x\ y)\ ((X'\ x\ y) :: (X'_s\ x\ y))).$$

Then we unify t and l', and get the most general unifier

$$\{F' \mapsto y_1, y_2.G,\ X' \mapsto \lambda y_1, y_2.y_1,\ X'_s \mapsto \lambda y_1, y_2.(y_2 :: nil)\}.$$

So,

$$t \leadsto \lambda x, y.\ ((G\ x) :: (map\ G\ (y :: nil))),$$

that is,

$$\{(t, s)\} \Longrightarrow_{(\text{Narrowing})} \{(\lambda x, y.\ ((G\ x) :: (map\ G\ (y :: nil))), s)\}.$$

Now we can use rule (Decomposition), and get the following new matching problem:

$$(1) \quad \{(\lambda x, y.G\ x, \lambda x, y.succ\ x), (\lambda x, y.\ (map\ G\ (y :: nil)), \lambda x, y.((succ\ y) :: nil))\}.$$

Because both $\lambda x, y.(G\ x)$ and $\lambda x, y.(succ\ x)$ are patterns containing no matching combinators, their most general unifier exists and is the following:

$$\{G \mapsto \lambda z.succ\ z\}.$$

Then, we have

$$\begin{array}{rl} (1) \Longrightarrow_{(\text{Matching})} & \{(G, \lambda z.succ\ z), \\ & (\lambda x, y.\ (map\ (\lambda z.succ\ z)\ (y :: nil)), \lambda x, y.((succ\ y) :: nil))\}. \end{array}$$

Since $\lambda x, y.\ (map\ (\lambda z.succ\ z)\ (y :: nil)) =_{\mathcal{E}_{map}} \lambda x, y.((succ\ y) :: nil)$, so

$$\begin{array}{l} \{(G, \lambda z.succ\ z), (\lambda x, y.((succ\ y) :: nil), \lambda x, y.((succ\ y) :: nil)) \\ \Longrightarrow_{(\text{Remove})} \{(G, \lambda z.succ\ z)\}. \end{array}$$

The match of t and s is $\{G \mapsto \lambda z.succ\ z\}$.

In order to show the application of (Deletion), we give a simple example. Let

$$\begin{array}{rcl} t & = & \lambda x.len((\lambda z.x + z) :: X_s) \\ s & = & \lambda x.succ\ 0 \end{array}$$

be two terms to be matched, where the function *len* defines the length of lists and can be found in section 2.1. After abstraction, we have

$$\{(\lambda x.len((\lambda z.F\ x\ z) :: X_s),\ \lambda x.succ\ 0), (\lambda x, z.x + z,\ \lambda x, z.F\ x\ z)\}.$$

This problem can be transformed to

$$\{(X_s, nil), (\lambda x, z.x + z, \lambda x, z.F\ x\ z)\}.$$

F does not occur anywhere else, so the second matching pair can be deleted. Such kind of matching pairs are produced by using the rule (Abstraction) to matching pairs, in which there are functions containing variable-dropping equations like *len*.

4.3.3 The Matching Algorithm $\mathcal{MA}^2$

The matching algorithm $\mathcal{MA}^2$ is similar to $\mathcal{MA}^1$. The central idea of the algorithm is to construct a matching tree (BUILD-TREE) for any given matching problem Γ using SIMPL, HO-MATCH, and NARROW. Each node is labeled by a matching problem, the root is Γ. If there is a node Γ' that is not in solved form, first simplify it using SIMPL, i.e., apply the rules (Remove), (Deletion), (Decomposition), and (Abstraction) exhaustively. Then choose a matching pair. If the pair consists only of pure terms, solve it using HO-MATCH, and get a set of matches, for each match σ draw an edge from Γ' to $\sigma\Gamma'$. Otherwise, apply all possible structural narrowing; we will get a set of new matching problems by NARROW, for each of them draw an edge from Γ' to it. Repeat this process, until all nodes are in solved form or no rule can be applied. See figure 9 about its structure.

The techniques applied to improve the algorithm $\mathcal{MA}^1$ can also be used to improve $\mathcal{MA}^2$, especially dependency analysis. In the higher-order case, only free variables will be considered. A matching pair (s, t) is called **dependent on** pair (s', t') if $\mathcal{FV}(t) \cap \mathcal{FV}(s') \neq \emptyset$, denoted also as $(s, t) \prec (s', t')$. A matching pair is called **solvable**, if it does not depend on any other pair. After simplifying a matching problem using SIMPL, we always choose a solvable matching pair, solve it using HO-MATCH, or apply all possible structural narrowing to it and solve the new matching problems using MATCH. After dependency analysis, matching trees will be easily maintained, useless narrowing steps are avoided, and each variable in a matching problem is instantiated only once.

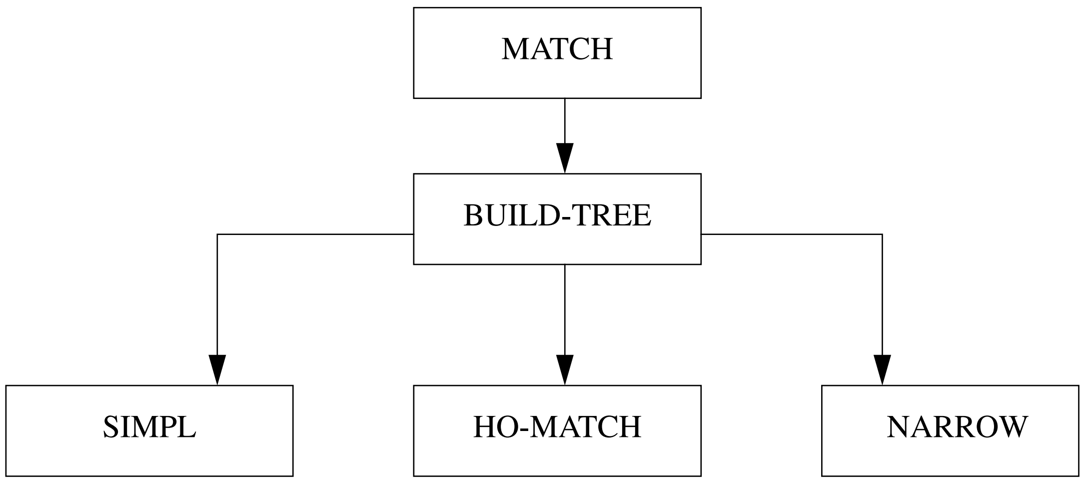

Figure 9: The structure of $\mathcal{MA}^2$

4.4 Correctness of the Matching Algorithm

In section 2.4, we have proved that any first-order matching problem, in which all functions are acyclic matching combinators, is decidable. The algorithm $\mathcal{MA}^1$ produces a finite complete set of matches for this class of matching problems. Fortunately, this does also hold for the higher-order case under the assumption that the class of pure higher-order matching problems concerned is decidable.

Theorem 4.6 Let Γ be a higher-order matching problem in which all functions are acyclic matching combinators, then the matching algorithm $\mathcal{MA}^2$ terminates with a complete set of minimal matches of Γ. □

This theorem presents four important results: **soundness**, **completeness**, **minimality** and **termination**. In the sequel, we consider only matching problems of higher-order constructor-based specifications, in which all functions are matching combinators. Again, we will concentrate on the transformation system $\mathcal{TS}^2$, since it is the basis of the matching algorithm $\mathcal{MA}^2$.

Theorem 4.7 (Soundness) Let Γ be a higher-order matching problem, $\Gamma \Longrightarrow^*_{\mathcal{TS}^2} \Gamma'$ such that Γ' is in solved form, then $\sigma_{\Gamma'}[\mathcal{FV}(\Gamma)]$ is a match of Γ. □

Theorem 4.8 (Completeness) Let Γ be a higher-order matching problem, σ a match of Γ, then there exists a transformation sequence $\Gamma \Longrightarrow^*_{\mathcal{TS}^2} \Gamma'$ with Γ' in solved form, such that $\sigma_{\Gamma'} \leq_{[\mathcal{FV}(\Gamma)]} \sigma$. □

These two theorems can be proved similarly to the proof of the soundness and completeness of the system $\mathcal{TS}^1$. We will not present the proofs here again. The following theorem is important in the proof.

Theorem 4.9 Let Γ be a higher-order matching problem, $\Gamma \Longrightarrow^*_{\mathcal{TS}^2} \Gamma'$, Γ' contains at least one solvable matching pair as long as Γ' is not in solved form.

Proof. If we can prove that the dependency relation $\prec$ is well-founded on Γ', then there is a set of pairs that do not depend on any other pair, since Γ' contains only finitely many matching pairs. Because no pair depends on solved pairs, some of them should be solvable matching pairs, as long as Γ' is not in solved form. Using induction on transformation sequences, the theorem can be proved.

Now we prove that $\prec$ is well-founded on Γ'. Let

$$\Gamma_1 \Longrightarrow_{\mathcal{TS}^2} \Gamma_2,$$

be a step of transformation, and $\prec$ be well-founded on Γ_1. To prove $\prec$ is well-founded on Γ_2, we consider the following cases:

- If the rule applied is (Remove), (Deletion), or (Decomposition), then $\prec$ is obviously well-founded on Γ_2, since these rules do not introduce new dependency relations between matching pairs.

- If rule (Abstraction) is applied to (s,t), some new pairs

$$(s_i, \lambda y_{i,1}, \cdots, y_{i,n_i}.F_i\ y_{i,1} \cdots y_{i,n_i})$$

  $(1 \leq i \leq k)$ will be introduced. Since F_i is a new free variable, thses newly introduced pairs are dependent only on the pair (s,t), so $\prec$ is also well-founded on Γ_2.

- If the rule applied is (Matching) or (Narrowing), some dependency relations will be removed. Lifting the lemmas 2.4 and 2.5 in section 2.3 to the higher-order case, we can show that no new relations will be introduced. □

Theorem 4.10 (Minimality) Let Γ be a higher-order matching problem, $\Gamma \Longrightarrow^*_{\mathcal{TS}^2} \Gamma'$ with Γ' in solved form, then Γ' forms a minimal match of Γ.

Proof. We present an outline hier. The only two rules to be considered are (Matching) and (Narrowing). The application of any other rule does not influence the matches.

- Let (t,s) be a solvable matching pair in a matching problem and t a pure higher-order term. Then, according to the assumption, one of the minimal matches of (t,s) will be returned after applying (Matching). That is, this rule always produces a minimal partial solution of a matching problem.

- Let $(\lambda x_1, \cdots, x_k.t, \lambda x_1, \cdots, x_k.s)$ be a solvable matching pair in a matching problem and equations $l_1 = r_1$ and $l_2 = r_2$ can be used to narrow it. If $\sigma_1 = mgu(\lambda x_1, \cdots, x_k.t, \lambda x_1, \cdots, x_k.\theta_1 l_1)$, and $\sigma_2 = mgu(\lambda x_1, \cdots, x_k.t, \lambda x_1, \cdots, x_k.\theta_2 l_2)$, where θ_1 and θ_2 are $\{x_1, \cdots, x_k\}$-lifter of l_1 and l_2 respectively, then $\sigma_1 \not\leq_{[\mathcal{FV}(t)]} \sigma_2$. Otherwise, there exists θ such that $(\theta \circ \sigma_1)(\theta_1 l_1) = \sigma_2(\theta_2 l_2)$, which contradicts the fact that any higher-order constructor-based specification is non-overlapping. □

Theorem 4.11 (Termination) If Γ is a higher-order matching problem, in which all functions are acyclic matching combinators, then any transformation sequence beginning with Γ terminates.

Similar to the first-order case, the central point of the proof is to define a well-founded ordering over higher-order matching problems. The definition of $|t|$ denoting the depth of matching combinators in t will be extended as:

$$\begin{array}{lcll}
|\lambda x.s| & = & |s| & \\
|a| & = & 0, & \text{if } a \text{ is a variable or a constructor} \\
|c\ t_1 \cdots t_n| & = & \max_{1 \leq i \leq n} |t_i|, & \text{if } c \text{ is a constructor or a variable} \\
|f\ t_1 \cdots t_n| & = & \max_{1 \leq i \leq n} |t_i|\ +\ 1, & \text{if } f \text{ is a matching combinator}
\end{array}$$

Let P be a suitable property along with the well-founded partial-ordering $\rhd$. $(s,t) \rhd_p (s',t')$ is defined similarly to the first-order case: $P(t) \rhd P(t')$, or $P(t) \sim P(t')$ and $|s'| < |s|$. So, $\rhd_p$ is a well-founded partial-ordering on higher-order matching pairs.

The multiset ordering $\rhd_\Gamma$ of $\rhd_p$ over matching problems can also be used for higher-order matching problems: Let Γ_1 and Γ_2 be two matching problems, then $\Gamma_1 \rhd_\Gamma \Gamma_2$ iff for any non-solved pair $(t_1, s_1) \in \Gamma_1 - \Gamma_2$, there exists a non-solved pair $(t_2, s_2) \in \Gamma_2 - \Gamma_1$ such that $(t_1, s_1) \rhd_p (t_2, s_2)$.

Proof (Theorem 4.11). Let

$$\Gamma_1 \Longrightarrow^*_{\mathcal{TR}^2} \Gamma' \Longrightarrow_{(\text{Narrowing})} \Gamma'' \Longrightarrow^*_{\mathcal{TR}^2} \Gamma_2$$

be a sequence of transformations in $\mathcal{TS}^2$. If we can show that $\Gamma_1 \rhd_\Gamma \Gamma_2$, then the theorem can be proved using the fact that $\rhd_\Gamma$ is a well-founded ordering, and all other rules can be used on a given matching problem only finitely many times.

Without loss of generality, we assume that Γ_1 contains only one matching pair (s, t) such that

- s has the form $\lambda x_1, \cdots, x_k.(f\ s_1 \cdots s_n)$,
- f is an acyclic matching combinator,
- t is a ground term, otherwise we can treat the variables in t as constants, and
- s contains no defined function nested below any proper decreasing function.

$$\begin{array}{ll} \{(s,t)\} & \\ \Longrightarrow^*_{(\text{Abstraction})} & \{(\lambda x_1, \cdots, x_k.(f\ t_1 \cdots t_n), t), (u_1, v_1), \cdots, (u_m, v_m)\} \\ \Longrightarrow_{(\text{Narrowing})} & \Gamma_\sigma \cup \{(s', t), (\sigma u_1, \sigma v_1), \cdots, (\sigma u_m, \sigma v_m)\} \end{array}$$

where $\lambda x_1, \cdots, x_k.(f\ t_1 \cdots t_n)$ is structurally narrowable to s' with an equation $l = r$ and the substitution σ. We show first that $(s, t) \rhd_p (s', t)$. There are two cases to be considered:

- The head of s' is a free variable. Because r is an extended pattern, s' is a pure term. That is, (s', t) can be solved by (Matching).
- The head of s' is a bound variable or a constructor. Then, rule (Decomposition) can be applied and transforms the matching pair to a set of pairs: $\{(r_1, t_1), \cdots, (r_m, t_m)\}$, and $(s', t) \rhd_p (r_i, t_i)$, since $P(t) \rhd P(t_i)$, for all $1 \le i \le m$.

Thus, by applying the induction hypothesis we can assume that all minimal matches for (s',t) can be found in finite time. Let θ be one of these matches, we have:

$$\theta(\sigma(\lambda x_1, \cdots, x_k.(f\ t_1 \cdots t_n))) =_{\mathcal{E}_f} \theta s' =_{\mathcal{E}_f} t.$$

Then, we show that

$$(s, t)\ \rhd_p\ ((\theta \circ \sigma) u_i, (\theta \circ \sigma) v_i),$$

for all $1 \le i \le m$. The following two cases should be considered:

- $((\theta \circ \sigma) u_i, (\theta \circ \sigma) v_i)$ is a pure higher-order matching pair, it can be solved by (Matching).
- Otherwise, f should be a non-decreasing function. So, $P(t) \rhd P((\theta \circ \sigma) v_i)$ or $P(t) \sim P((\theta \circ \sigma) v_i)$, according to the definition of proper non-decreasing functions and the fact that $\theta \circ \sigma$ is a most general match of $\lambda x_1, \cdots, x_k.(f\ t_1 \cdots t_n)$ and t. Additionally, $|(\theta \circ \sigma) u_i| < |s|$, since for any X, $\sigma(X)$ contains no matching combinator. □

Combining all these theorems, theorem 4.6 can be proved.

4.5 Applications

In this section, we first discuss higher-order contextual matching. The applications of extended higher-order matching to program transformations will be shown through examples.

4.5.1 Higher-Order Contextual Matching

In chapter 3 we have discussed contextual matching for first-order matching problems. The goal of contextual matching is to reduce the number of matches, and consequently to speed up the matching process. For the higher-order case, this is even more important, because the number of matches of a higher-order matching problem may be very large through combining higher-order matching with semantic matching. Recall that the three basic steps to compute first-order contexts are:

- Simplification through term rewriting.
- Instantiation through substitution.
- Verification through constant equivalence and non-equivalence.

To introduce contexts into higher-order matching problems, we should first study higher-order rewriting in more detail. In Nipkow's paper [59], it is proved that orthogonal higher-order rewriting systems are confluent. We do not know yet whether this result can be extended to more general rewriting systems. In [69], an algebraic technique is presented to prove the termination of higher-order rewriting systems. We assume that all functions used in contexts are defined by confluent and terminating higher-order specifications. Since the left-hand side of each higher-order equation is a pattern, if a term can be rewritten using such an equation, the most general match can always be computed through higher-order pattern unification.

The substitutions about λ-terms are introduced in the first section of this chapter. This is quite standard, and there is no need to discuss them again.

To verify higher-order contexts, we should extend the definitions of constant equivalence and non-equivalence.

Definition 4.11 Let t and s be two normal forms. t and s are called **higher-order constant equivalent** (shortly, **constant equivalent**), denoted as $t \stackrel{c}{=} s$, iff

- $t = \lambda x_1, \cdots, x_k.t'$, $\lambda x_1, \cdots, x_k.s'$, t' and s' are the same constant, or the same bound variable; or
- $t = \lambda x_1, \cdots, x_k.(f\ t_1 \cdots t_n)$, $s = \lambda x_1, \cdots, x_k.(f\ s_1 \cdots\ s_n)$, and $\lambda x_1, \cdots, x_k.t_i \stackrel{c}{=} \lambda x_1, \cdots, x_k.s_i$ for all $1 \leq i \leq n$, where f is an n-arity $(n > 0)$ constructor, function symbol, or bound variable.

t and s are called **higher-order constant non-equivalent** (shortly, **constant non-equivalent**), denoted as $t \stackrel{c}{\neq} s$, iff

- t and s are ground terms, but t and s are not constant equivalent; or

- $t = \lambda x_1, \cdots, x_k.(c_1\ t_1 \cdots\ t_n)$, $s = \lambda x_1, \cdots, x_k.(c_2\ s_1 \cdots\ s_n)$, and c_1 and c_2 are different constructor symbols, or different bound variables.

Under these considerations, higher-order contextual matching problems can be solved by the matching algorithm $\mathcal{MA}^2$ with the same extensions as in the first-order case.

4.5.2 Higher-Order Matching

Elimination of common subexpressions

The elimination of common subexpressions of an expression is an often used optimization method in program transformations. The associated transformation rule can be presented as

$$F\ X \implies \text{LET } Y = X \text{ IN } F\ Y,$$

supposing that $F\ X$ is a second-order term and both F and X are free variables. To make the rule meaningful, we require that an expression substituted for X is not a subexpression of the expression substituted for F. The following is an example specification that defines the subexpression relation. Let *Var* be a specification defining the sort *Var* and the function *eqV*: $var \to var \to bool$; *Con* defines the sort *con* and the function *eqC*: $con \to con \to bool$.

$$\begin{array}{rl}
Exp = & Bool\ +\ Var\ +\ Con\ + \\
\textbf{sorts}: & exp \\
\textbf{cons}: & v\ :\ var \to exp \\
& c\ :\ con \to exp \\
& ap\ :\ exp \to exp \to exp \\
& abs\ :\ var \to exp \to exp \\
\textbf{funs}: & eqExp,\ subExp\ :\ exp \to exp \to bool \\
\textbf{eqns}: & \forall\ t, t_1, t_2 \in exp;\ x \in var;\ y \in con: \\
& \ldots \\
& subExp\ t\ (v\ x) = eqExp\ t\ (v\ x) \\
& subExp\ t\ (c\ y) = eqExp\ t\ (c\ y) \\
& subExp\ t\ (ap\ t_1\ t_2) = or\ (eqExp\ t\ (ap\ t_1\ t_2)) \\
& \qquad\qquad (or\ (subExp\ t\ t_1)\ (subExp\ t\ t_2)) \\
& subExp\ t\ (abs\ x\ t_1) = and\ (not\ (eqExp\ t\ (v\ x))) \\
& \qquad\qquad (or\ (eqExp\ t\ (abs\ x\ t_1))\ (subExp\ t\ t_1))
\end{array}$$

where the function *eqExp* proves whether two terms are equal or not using *eqV* and *eqC*; its definition is omitted here.

Let $(a+b)*(a+b)$ be an expression, where $+$ and $*$ are two constructors (not matching combinators), a and b are two expressions. To eliminate common expressions in it, we should solve the following higher-order contextual matching problem:

$$\{(F\ X, (a+b)*(a+b))\}\ [EQ(subExp\ X\ F, false)].$$

Using the second-order matching algorithm in [40], we will get more than ten matches, but only the following four satisfy the condition:

$$\{F \mapsto \lambda x.x, X \mapsto (a{+}b){*}(a{+}b)\},$$
$$\{F \mapsto \lambda x.(a{+}x){*}(a{+}x), X \mapsto b\},$$
$$\{F \mapsto \lambda x.(x{+}b){*}(x{+}b), X \mapsto a\},$$
$$\{F \mapsto \lambda x.x{*}x, X \mapsto a{+}b\}.$$

The associated transformations are

$$\begin{aligned}
(a{+}b){*}(a{+}b) &\Longrightarrow \text{LET } Y = (a{+}b){*}(a{+}b) \text{ IN } Y,\\
(a{+}b){*}(a{+}b) &\Longrightarrow \text{LET } Y = b \text{ IN } (a{+}Y){*}(a{+}Y),\\
(a{+}b){*}(a{+}b) &\Longrightarrow \text{LET } Y = a \text{ IN } (Y{+}b){*}(Y{+}b),\\
(a{+}b){*}(a{+}b) &\Longrightarrow \text{LET } Y = a{+}b \text{ IN } Y{*}Y.
\end{aligned}$$

Of course, the last transformation is the best. To get a best match, generally we need to compare all matches produced. This is beyond our concern here.

Transformation of linear recursion to iteration

The transformation from recursion to iteration is a fundamental concept. The basic schema of a linear recursion is (see [5])

$$f =_{def} \lambda x.\ (\text{IF } B\ X \text{ THEN } f(H\ x) \oplus (K\ x) \text{ ELSE } T\ x),$$

which defines the function f linearly recursively. Under the condition that $\oplus$ is associative and n is the right identity of $\oplus$, it can be transformed to the following iterative schema:

$$\begin{aligned}
&\lambda x.f\ x = g\ x\ n\\
&\text{WHERE } g =_{def} \lambda x, y.\ (\text{IF } B\ X \text{ THEN } g\ (H\ x)\ ((K\ x) \oplus y) \text{ ELSE } (T\ x) \oplus y).
\end{aligned}$$

For instance, we want to transform the function *fac* defined by the following linear recursion:

$$fac =_{def} \lambda x.\ (\text{IF } x{>}\ 0 \text{ THEN } fac(x\text{-}1){*}x \text{ ELSE } 1).$$

Again -, *, and > are constructors. The matching problem to be solved is

$$\{(\lambda x.B\ x, \lambda x.x{>}\ 0), (\lambda x.fac(H\ x){*}(K\ x), \lambda x.fac(x\text{-}1){*}x), (\lambda x.T\ x, 1)\}.$$

Note that f and $\oplus$ in the recursive schema are instantiated with *fac* and * respectively. The first and the third matching pairs contain only patterns. Solving them we get

$$\{B \mapsto \lambda x.x{>}\ 0, T \mapsto \lambda x.1\}.$$

The second matching pair can be decomposed to:

$$\{(\lambda x.fac(H\ x), \lambda x.fac(x\text{-}1)), (\lambda x.K\ x, \lambda x.x)\}.$$

It contains only patterns too; the most general match is

$$\{H \mapsto \lambda x.x\text{-}1,\ K \mapsto \lambda x.x\}.$$

Assume * is associative and 1 is its right identity, then the iterative version of *fac* is

$$\begin{aligned}
&\lambda x.fac\ x = g\ x\ 1\\
&\text{WHERE } g =_{def} \lambda x, y.\ (\text{IF } x{>}\ 0 \text{ THEN } g\ (x\text{-}1)\ (x{*}y) \text{ ELSE } y).
\end{aligned}$$

This example also shows that a lot of higher-order matching problems are in fact matching problems of patterns in practice.

4.5.3 Higher-Order Matching Combinators

List manipulation

In program transformations, the manipulation of lists occurs quite often, for example, to transform a declaration of a list of variables to a list of declarations. The higher-order function *seq* defined in the following specification is very helpful in such situations.

$$
\begin{array}{rll}
Seq[\alpha] & = & Nat \;+\; List[\alpha] \;+ \\
& \textbf{funs}: & proj \;:\; nat \rightarrow list \rightarrow \alpha \\
& & seq \;:\; nat \rightarrow (nat \rightarrow \alpha) \rightarrow list \\
& \textbf{eqns}: & \forall\; N \in nat;\; X \in \alpha;\; X_s \in list;\; F \in nat \rightarrow \alpha : \\
& & proj\,(succ\;0)\;(X :: X_s) = X \\
& & proj\,(succ(succ\;N))\;(X :: X_s) = proj\,(succ\;N)\;X_s \\
& & \\
& & seq\;0\;F = nil \\
& & seq\,(succ\;N)\;F = (seq\;N\;F) +\!\!+ (F(succ\;N) :: nil)
\end{array}
$$

Now the transformation of "putting a 'context' of a list into each of its elements" can be presented as the following rule:

$$C(seq\;N\;(\lambda i.(proj\;i\;X_s))) \;\Longrightarrow\; seq\;N\;(\lambda i.C(proj\;i\;X_s)).$$

Let $a, b, c : t$ be a declaration to be transformed by the rule, where : is again a constructor, a, b and c are variables, and t is a type in the program where the declaration occurs. The matching problem is

$$\{(C(seq\;N\;(\lambda i.(proj\;i\;X_s))), [a,b,c]{:}t)\},$$

where '[,]' is used to present a list. Of course, this is a higher-order semantic matching problem. The following is a match:

$$\{N \mapsto succ(succ(succ\;0)),\; X_s \mapsto [a,b,c],\; C \mapsto \lambda x.x{:}t\}.$$

So

$$[a,b,c]{:}t \;\Longrightarrow\; [a{:}t,\;\; b{:}t,\;\; c{:}t].$$

It is very important to provide the concrete function $\lambda i.(proj\;i\;X_s)$ in the transformation rule. Otherwise, we may have the unsolvable matching pair $(seq\;N\;F,\;[a,b,c])$, since the second-order matching problem

$$\{([F(succ\;0), F(succ(succ\;0)), F(succ(succ(succ\;0)))], [a,b,c])\}$$

is unsolvable.

In fact, $seq\;N\;(\lambda i.(proj\;i\;X_s))$ equals to X_s if the length of X_s is N. For different concrete functions, we will have different transformation rules using *seq*.

$$C(seq\;N\;(\lambda i.(proj\,(succ(N-i))\;X_s))) \;\Longrightarrow\; seq\;N\;(\lambda i.C(proj\;i\;X_s)).$$

With this transformation rule we can put the context of a list into each of its element and reverse the list at the same time, since $seq\;N\;(\lambda i.(proj\,(succ(N-i))\;X_s)$ equals to the reverse list of X_s if the length of X_s is N.

4.6 Related Work

There are several different definitions concerning higher-order term rewriting systems. In this chapter we have considered the definition from [58, 59].

In [46], Klop defined combinatory reduction systems, which provide a framework for term rewriting incorporating the usual term rewriting systems and also systems like λ-calculus plus combinatory logic. For instance, the following β-reduction rule forms a combinatory reduction system:

$$(\lambda x.F\ x)Y \rightarrow F\ Y.$$

If F is replaced with $\lambda x.G\ x\ x$ and Y with $a\ b$, then we have the following rewriting step:

$$(\lambda x.G\ x\ x)\ (a\ b) \rightarrow G\ (a\ b)\ (a\ b).$$

If $l \rightarrow r$ is a rewrite rule of a combinatory rewrite system, then

- the leftmost symbol of l is a constant, such as λ in the above example;
- $\mathcal{FV}(r) \subseteq \mathcal{FV}(l)$;
- every free variable F in l occurs only in the form $F\ x_1 \cdots x_k$, where the x_i $(1 \leq i \leq k)$ form a list of distinct bound variables.

A formal definition of combinatory reduction systems can be found in [46], which is a generalization of the definition we used.

In [84], Wolfram gave another definition of higher-order rewriting systems: A higher-order rewrite rule has the form

$$\lambda x_1, \cdots, x_n.s \rightarrow \lambda x_1, \cdots, x_n.t$$

where both sides are ground terms and in long $\beta\eta$-normal form. Let u and v be two ground terms of the same type in long $\beta\eta$-normal form. $u \rightarrow v$, if there exist a rewrite rule $\lambda x_1, \cdots, x_n.s \rightarrow \lambda x_1, \cdots, x_n.t$, a substitution σ, and a position p in u such that

- $u|_p = \lambda y_1, \cdots y_k.u'$ and u' and s have the same type;
- $(\sigma s)\updownarrow = u'$;
- $v = (u[\lambda y_1, \cdots y_k.\sigma t]_p)\updownarrow$.

This definition is also a generalization of the one we used. The left-hand side of a higher-order rewrite rule is not necessarily a pattern, so there may exist several most general substitutions σ such that $(\sigma s)\updownarrow = u'$.

Matching problems associated with higher-order constructor-based specifications are a special class of higher-order semantic matching problems. In this chapter, we have developed a concrete matching algorithm to solve this class of matching problems and proved its completeness. Furthermore, the algorithm terminates if we consider only acyclic matching combinators.

To solve the higher-order semantic matching problems, we have met the following challenges:

- To get a set of practically useful higher-order specifications, we have extended the higher-order patterns to extended patterns. They cover more practically interesting λ-terms, but can be easily transformed to higher-order patterns.

- To ensure the decidability of higher-order semantic matching problems, we have generalized the definition of non-decreasing functions to higher-order proper non-decreasing functions. So we can treat functions with variable-dropping equations, which is particularly important to the higher-order case.

- To make the implementation efficient and easy, we have lifted structural narrowing to the higher-order case and proved that it is complete together with pattern abstraction and function abstraction.

The work of Miller [55], Nipkow [58, 60, 59], Huet and Lang [40], Middeldorp and Hamoen [54], Qian [71], and the decidable first-order semantic matching discussed in chapter 2 and 3 provide the important prerequisites for this work.

There is some work about higher-order unification modulo first-order term rewriting systems. In [71], Qian lifted the completeness result of first-order narrowing strategies to patterns for first-order rules. There is only little work has been done about higher-order unification with respect to higher-order term rewriting systems, let along the research about the decidability of higher-order semantic unification and matching. In the rest of this section, we will present the work in [1] and [70], one by Avenhaus and Loría-Sáenz, the other by Prehofer.

Solving higher-order equations

To the best of our knowledge, Prehofer's work [70] gives the first framework for solving higher-order equations by several higher-order narrowing approaches. In that paper, it is proved that the general first-order notion of narrowing can be lifted to higher-order patterns; at the same time the completeness of first-order narrowing strategies is maintained. For the full higher-order case, it seems less promising. Among several technical problems, handling locally bound variables in the solution is a main problem. With the alternative of lazy narrowing (see below), most of these problems can be avoided by integrating equational reasoning into unification. A system called NL is presented, which essentially consists of the transformation rules for higher-order unification [80] plus the following two lazy narrowing rules:

(**Lazy Narrowing**)

$$\begin{aligned} & \{(\lambda x_1, \cdots, x_k.(f\ s_1 \cdots s_n), \lambda x_1, \cdots, x_k.t)\} \cup \Gamma \\ \Longrightarrow\ & \{(\lambda x_1, \cdots, x_k.s_1, \lambda x_1, \cdots, x_k.l_1), \cdots, (\lambda x_1, \cdots, x_k.s_n, \lambda x_1, \cdots, x_k.l_n), \\ & (\lambda x_1, \cdots, x_k.r, \lambda x_1, \cdots, x_k.t)\} \cup \Gamma, \end{aligned}$$

where $f\ l_1 \cdots l_n \to r$ is an $\{x_1 \cdots, x_k\}$-lifted rule.

(**Lazy Narrowing at Variable**)

$$\begin{aligned} & \{(\lambda x_1, \cdots, x_k.(F\ s_1 \cdots s_n), \lambda x_1, \cdots, x_k.t)\} \cup \Gamma \\ & \Longrightarrow \{(\lambda x_1, \cdots, x_k.(F\ s_1 \cdots s_n), \lambda x_1, \cdots, x_k.l), (\lambda x_1, \cdots, x_k.r, \lambda x_1, \cdots, x_k.t)\} \cup \Gamma, \end{aligned}$$

where $l \to r$ is an $\{x_1 \cdots, x_k\}$-lifted rule.

Our system $\mathcal{TS}^2$ solves constructor-based higher-order specifications which are of course not so general as the rewriting systems discussed in [70]. However, we can use structural narrowing, together with pattern abstraction and combinator abstraction, to solve higher-order equations, and our matching algorithm is more concrete and efficient for implementation. Furthermore, our algorithm always produces a set of minimal matches and terminates for a class of acyclic higher-order matching combinators.

A promising approach suggested in Prehofer's paper is to divide a higher-order semantic matching problem into a pattern part, where narrowing works as in the first-order case, and a non-pattern part. Pattern abstraction defined for extended patterns is similar to this idea, it is a concrete solution to this problem.

Higher-order conditional narrowing

Recently, Avenhaus and Loría-Sáenz presented a paper about higher-order conditional rewriting and narrowing [1]. Higher-order conditional term rewriting systems combine the first-order conditional rewriting systems introduced by Kaplan [45] and higher-order unconditional rewriting systems introduced by Nipkow [58]. The method used to treat conditions of higher-order rewriting rules in rewriting and narrowing is similar to the method for the first-order case [45].

A large part of the paper copes with the following problems:

- To give a sufficient condition for the confluence and termination of higher-order conditional term rewriting systems.

- To define a special class of higher-order terms such that in any narrowing derivation, if term t narrows to t' with a rule $l \to r \Leftarrow c$ at position p, then $t|_p$ is a higher-order pattern, so that the most general unifier exists.

Under these conditions, it is proved that higher-order conditional narrowing is complete.

Chapter 5

Implementation Issues

In general, there are two approaches for the implementation of a unification or a matching algorithm: using a higher-order language or using a low-level abstract machine. Early implementations of narrowing, like the RAP system [25], NARROWER [73], functional logic languages, such as SLOG [22] that is based on normalizing innermost narrowing, were all implemented in higher-level languages like Pascal, Ada or LISP. The higher-order unification algorithm used in the Isabelle system [63] was implemented in the functional language ML. The use of "abstract machines" is also a well-known technique for the efficient implementation of functional logic programming languages on standard hardware. For instance, the "Warren Abstract Machine" (WAM) [2], which is the basis of most Prolog implementations and supports logical variables, unification, and so on, and the K-LEAF system [9], which supports a lazy narrowing strategy, were implemented on low-level machines.

Our goal is to provide a prototyping system for program transformations, which will be supported by the proof system Isabelle, so we have chosen the first approach and implemented the matching algorithm in ML.

5.1 Extended Higher-Order Abstract Syntax

Extended matching discussed in the previous chapters supports a powerful notion of elementary program transformations. Extended matching is in fact generic with respect to concrete object languages, so it can be used as a framework to support transformations of programs in any object language or between different languages. The implementation of the matching algorithm in the framework of a transformation system thus provides a formal tool for elementary transformations.

There are some systems that use the λ-calculus to represent logical formulas and programs, and use deduction systems or higher-order unifications for manipulating them. LF, the Edinburgh Logical Framework [32], makes explicit use of a version of the λ-calculus, and a deduction system to support the reasoning. Since LF uses dependent function types, it is not known whether there is a unification algorithm that could be used for proof search in the system in a general way.

The *higher-order abstract syntax* from Pfenning and Elliott [66] uses simply typed λ-calculus enriched with products and polymorphism as a central representation for programs, formulas, rules, and other syntactic objects in program manipulation and other formal systems, where matching, substitution, and unification are central operations. The most important advantage of higher-order abstract syntax is to incorporate name binding information in a uniform and language generic way. Higher-order abstract syntax, a supporting matching and unification algorithm have been implemented in Common Lisp in the framework of the Ergo project at Carnegie Mellon University.

Similar to higher-order abstract syntax, Isabelle [63] also uses a higher-order abstract syntax, and provides higher-order unification as a central operation. Isabelle's λ-calculus does not have the product as in the higher-order abstract syntax in [66], but can explicitly encode quantifier dependencies.

Inspired by these ideas, we also use a higher-order abstract syntax to represent programs and to implement the extended matching algorithm $\mathcal{MA}^2$ to support program transformations. The abstract syntax we use enriches the simply typed λ-calculus with polymorphism and matching combinators and thus extends the higher-order abstract syntax of [66]. The introduction of combinators enables the presentation of complex program schemata, the abstraction of transformation rules, and the explicit expression of some context sensitive conditions.

The basic principles that guide the design of an internal representation for programs in our context are:

- The representation should support a simple syntactic and semantic definition of object languages.
- The representation should be general enough to be used by all relevant tools in an environment, thereby providing for a high degree of integration between the tools.
- The system must be able to correctly and efficiently transform programs.

5.1.1 Some Motivating Examples

In this section we highlight some of the problems that arise in matching and substitution during program transformations. The program transformation rules we present throughout this thesis are stated without any semantic side conditions such as strictness, termination, and so on. Depending on the object language semantics, such conditions may still be necessary to ensure full semantic equivalence between the transformed programs.

"Correct matching and substitution" is a familiar problem, which is generally called "variable capturing". It appears in two different forms: during matching and during substitution. Consider the following transformation rule:

$$\textsc{let}\ Y = A\ \textsc{in}\ B \implies B[A/Y], \qquad (1)$$

which eliminates a "LET-IN" expression. The following is an *incorrect* application of this transformation rule:

$$\begin{array}{ll} & \textsc{let}\ u = v\ \textsc{in}\ \textsc{let}\ v = 5\ \textsc{in}\ u * v \\ \implies & \textsc{let}\ v = 5\ \textsc{in}\ v * v \\ \implies & 5 * 5. \end{array}$$

What is required for correct substitution is the recognition of name conflicts and renaming of bound variables.

A similar problem is "variable occurrence restrictions" which again requires renaming of bound variables during substitution. This can be seen from the following transformation rule:

$$C(\text{IF } P \text{ THEN } S_1 \text{ ELSE } S_2) \implies \text{IF } P \text{ THEN } C\ S_1 \text{ ELSE } C\ S_2. \qquad (2)$$

Using the rule, we may have the *incorrect* transformation:

$$\begin{array}{ll} & \text{LET } b = \text{FALSE IN IF } b \text{ THEN } 1 \text{ ELSE } 2 \\ \implies & \text{IF } b \text{ THEN LET } b = \text{FALSE IN } 1 \\ & \text{ELSE LET } b = \text{FALSE IN } 2. \end{array}$$

Again, syntactic conditions on C are difficult to formulate if one wishes to eliminate the possibility of incorrect rule applications as in the above example. The use of higher-order abstract syntax solves this problem by allowing a rule as above, but automatically prohibiting the incorrect application during higher-order matching with some additional conditions.

The transformation rule (1) can also be represented as a conditional rule to avoid incorrect applications.

$$\begin{array}{l} \text{LET } Y = A \text{ IN } B \implies B[A/Y] \\ \Leftarrow \text{ each free variable in } A \text{ is not bound in } B \qquad (3) \end{array}$$

If this rule is used by matching against the left-hand side, the restriction on A must be considered as a context to be checked during the matching process.

5.1.2 Extended Higher-Order Abstract Syntax

Originally, a parser creates untyped abstract syntax trees. They can be viewed as terms that represent the syntactic structures of object programs. Variables do not exist in this representation, that is, object language variables are represented as identifiers, which are all considered as constants.

In the context of program manipulation tools, the need for types in the abstract syntax soon arises. This is because one needs to be able to test whether one substitution of a subterm for another is syntactically legal, or a transformation rule is type-preserving. The types are derived from the "syntactic sorts" or "syntactic categories" of an object language. The notions of syntactic sorts and object language types usually do not coincide, though typically object language types refine the structure of syntactic sorts.

Motivated by the examples above and by the semantic matching algorithm in the previous chapters, we define **extended higher-order abstract syntax** now, which is the simply typed λ-calculus enriched with polymorphism and matching combinators.

The functional language ML will be used to describe abstract syntax. Note that we refer to three different levels of languages:

- ML, which describes extended higher-order abstract syntax and object languages. ML syntactic objects will be written in typewriter style: e.g. `string` denotes the ML type "string"; `datatype` denotes the ML keyword "datatype".
- Extended higher-order abstract syntax, in which object languages will be represented.

- The abstract syntax of an object language, whose syntactic objects will be denoted in small caps style. For instance, LET and ITE denote "LET-IN" and "IF-THEN-ELSE" expressions respectively.

The functional language ML

ML [33] is a strongly typed functional programming language, which has been used by a number of people for serious work during the last several years. Here we only want to summarize some important points of the language, and give some syntactical definitions which will be used in this chapter. Any other (functional) language with similar properties would do just as well.

In functional programming, functions are first-class data objects, they may be passed as arguments, and returned as results. The principal control mechanism in ML is recursive function application. ML is strongly typed and supports polymorphic types. Every legal expression has a uniquely-determined most general type automatically inferred by the compiler. Strong typing guarantees that no program can produce type errors at run time.

In this chapter only a small set of ML-types is used to describe abstract syntax. In the following we list some built-in types.

- The type `int` is the set of positive and negative integers. Integers are written in the usual way.
- The type `string` consists of finite sequences of characters, which are written in the conventional fashion as characters between double quotes, such as "LET" and "5".
- The type `tuple`, written as $\tau_1 * \tau_2$, is the type of ordered pairs of the form (t_1, t_2), where t_1 has type τ_1 and t_2 has type τ_2. We can build ordered n-tuples ($n \geq 2$) by writing n comma-separated expressions between parenthesis.
- The type τ `list` consists of finite sequences of values of type τ. `nil` and `::` are two constructors of lists. A list can also be written as comma-separated expressions between brackets, for instance $[1, 2, 3]$ is an `int list` consisting of three integers.

The type system of ML is extendible. Three forms of type bindings are available, each introducing an identifier as a type constructor. The simplest form of type binding is the "type abbreviation". A type constructor is defined, perhaps with parameters, as an abbreviation for a type expression. The following type abbreviation

$$\texttt{type}\ \mathit{intpair} = \texttt{int * int}$$

defines a type constructor *intpair* without parameter. Notice that there is no difference between `int * int` and *intpair*, because *intpair* is defined to be equal to `int * int`. "`datatype`" binding is also a way to define new compound types. A datatype is specified by giving it a name (and perhaps some type parameters) and a set of constructors for building objects of that type. The built-in type τ `list` can be considered to have been defined as follows:

$$\texttt{datatype}\ 'a\ \texttt{list} = \texttt{nil} \mid \ :: \ \texttt{of}\ 'a * 'a\ \texttt{list},$$

where $'a$ is a type parameter. Another type binding is by abstract types, which will not be used in this chapter.

Types of extended higher-order abstract syntax

Types belong to the ML type *typ*, which is a concrete datatype with four constructors; there are four kinds of types.

$$
\begin{array}{lcl}
\texttt{datatype}\ typ & = & Ide\ \texttt{of string} \\
 & \mid & Var\ \texttt{of string} \\
 & \mid & Fun\ \texttt{of}\ typ * typ \\
 & \mid & Ap\ \texttt{of}\ typ * typ
\end{array}
$$

- $Ide(st)$ denotes a type whose name is st. The types *nat*, *bool* used in the previous chapters are the type identifiers: $Ide(\text{“}nat\text{”})$ and $Ide(\text{“}bool\text{”})$ respectively.
- $Var(st)$ is a type variable.
- $Fun(T_1, T_2)$ presents a function type from T_1 to T_2, e.g., $nat \rightarrow nat$ and $nat \rightarrow bool$ are represented as $Fun(Ide(\text{“}nat\text{”}),\ Ide(\text{“}nat\text{”}))$ and $Fun(Ide(\text{“}nat\text{”}),\ Ide(\text{“}bool\text{”}))$ respectively.
- $Ap(T_1, T_2)$ denotes a type application, e.g., the parameterized type *list* is then $Ap(Ide(\text{“}list\text{”}),\ Var(\text{“}\alpha\text{”}))$, and $Ap(Ide(\text{“}list\text{”}),\ Ide(\text{“}nat\text{”}))$ is an instance.

The introduction of parameterized types solves the problem of the presence of lists of various forms: lists of arguments, lists of declarations, lists of variables. The usual solution is to curry functions, which enables representations, but not a simple, finitary definition of transformations. Again, we have three different levels of types:

- ML types which are used to describe syntactic objects of the abstract syntax.
- Types of extended higher-order abstract syntax, in which syntactic sorts of an object language will be presented.
- Types of an object language, which refine the structure of syntactic sorts.

To simplify the notation of *typ*, we introduce the following abbreviations:

$$
\begin{array}{lll}
[t] & \text{for} & Ide(\text{“}t\text{”}) \\
['\alpha] & \text{for} & Var(\text{“}\alpha\text{”}) \\
[t_1 \rightarrow t_2] & \text{for} & Fun([t_1], [t_2]) \\
[t_1\ t_2] & \text{for} & Ap([t_1], [t_2])
\end{array}
$$

Thus, we write, e.g., $[list\ '\alpha]$ instead of $Ap(Ide(\text{“}list\text{”}),\ Var(\text{“}\alpha\text{”}))$.

Terms of extended higher-order abstract syntax

Terms belong to the ML type *term*, which is a concrete datatype with six kinds of terms, compatible with the terms used in the Isabelle system. The only extension is the introduction of combinators.

$$
\begin{array}{lcl}
\texttt{datatype}\ term & = & Const\ \texttt{of string}\ *\ typ \\
 & | & Comb\ \texttt{of string}\ *\ typ\ *\ (eq\ \texttt{list}) \\
 & | & Free\ \texttt{of string}\ *\ typ \\
 & | & Bound\ \texttt{of int} \\
 & | & Abs\ \texttt{of string}\ *\ typ\ *\ term \\
 & | & \$\ \texttt{of}\ term\ *\ term \\
 & & \\
\texttt{datatype}\ eq & = & Eq\ \texttt{of}\ term\ *\ term
\end{array}
$$

- $Const(st, T)$ is a constant with name st and type T, e.g. $Const(\text{“}0\text{”}, [nat])$ denotes the constant 0, $Const(\text{“}succ\text{”}, [nat \rightarrow nat])$ denotes the constructor *succ*.
- $Comb(st, T, eqs)$ presents a matching combinator with name st, type T, and a list of equations eqs defining it.
- $Free(st, T)$ is a free variable with name st and type T.
- $Bound(n)$ is a bound variable with de Bruijn index n (see below).
- $Abs(st, T, t)$ denotes an abstraction with body t, whose bound variable has name st and type T. The following term represent the identity $\lambda x.x$.

$$Abs(\text{“}x\text{”}, [{'\alpha}], Bound(0)).$$

- $t_1\ \$\ t_2$ denotes the application of t_1 to t_2.

The combinators make extended higher-order abstract syntax more powerful than the higher-order abstract syntax introduced by Pfenning and Elliott in [66]. The combinators are functions defined by a set of equations, which need not be matching combinators. They can also be functions used to form conditions of transformation rules.

Convention: The types in terms will be omitted, e.g., we write $Const(\text{“}succ\text{”})$ and $Abs(\text{“}x\text{”}, Bound(0))$, instead of $Const(\text{“}succ\text{”}, [nat \rightarrow nat])$ and $Abs\ (\text{“}x\text{”}, [{'\alpha}], Bound(0))$.

Name-free representation

To simplify β-reduction and to avoid α-conversion, de Bruijn discovered the **name-free representation** [12], where each bound variable refers to its nesting depth of abstractions, called **de Bruijn index**, giving the number of abstractions lying between it and its binding abstraction.

In the name-free notation, no variable name appears after the λ symbol and bound variables appear as numbers. The first occurrence of x in the body of $\lambda x.(\lambda y.x)x$ is represented by 1, because it is enclosed in an abstraction over y. The second occurrence of x is not enclosed in any other abstraction and is represented by 0. Therefore the name-free representation of the term is $\lambda.(\lambda.1)0$. The name-free representation of the following term

$$\lambda x.x(\lambda y.x\ y\ (\lambda z.x\ y\ z))$$

is $\lambda.0(\lambda.1\ 0\ (\lambda.2\ 1\ 0))$. Two important advantages of using the name-free notation are:

- Avoiding α-conversion, since two λ-terms are α-equivalent iff their name-free representations are identical.
- Simplifying β-reduction, since there is no need to rename bound variables. For instance, to reduce $(\lambda x.\lambda y.x\ y)y$, usually we should first rename the bound variable y to a new variable, then apply the β-reduction:

  $$(\lambda x.\lambda y.x\ y)y \rightarrow_{\alpha} (\lambda x.\lambda z.x\ z)y \rightarrow_{\beta} \lambda z.y\ z.$$

  Using the name-free notation, we have

  $$(\lambda.\lambda.1\ 0)y \rightarrow_{\beta} \lambda.y\ 0.$$

In our abstract syntax, the names of bound variables will be remembered as strings. The only reason is to retrieve the bound variables for concrete presentations.

5.1.3 Representation of Object Languages

We have already used some object language syntax to show examples of program transformations in the previous chapters. As an example, we will use a simple functional language with sequences of definitions, because it provides a good framework for illustrating the use of the higher-order feature and the use of combinators. We begin with a grammar, which is also described in ML syntax.

`type` *prog*	=	*decl* `list`
`datatype` *decl*	=	DECL `of` *pat* `*` *exp*
`type` *pat*	=	`string list`
`datatype` *exp*	=	VAR `of string`
	|	CON `of string`
	|	ITE `of` *exp* `*` *exp* `*` *exp*
	|	LET `of string` `*` *exp* `*` *exp*
	|	AP `of` *exp* `*` *exp*
	|	LAM `of string` `*` *exp*

- Here, we assume that both variables and constants are of `string`s.
- The left-hand side of a declaration (*decl*) is a pattern of a list of variables.

Now we have both, extended higher-order abstract syntax and the abstract syntax of the object language. For simplicity, we call the former **meta-syntax** and the latter **object-syntax**. The next step we should do is to express the object-syntax in the meta-syntax.

In the object-syntax, there are four syntactic sorts: *exp*, *pat*, *decl*, and *prog*. We treat them as the following simple types in the meta-syntax:

$$\begin{array}{l} [exp] \\ [list\ \texttt{string}] \\ [decl] \\ [list\ decl] \end{array}$$

Let t be any syntactic object of the object-syntax; then its representation in the meta-syntax, denoted as $\langle t \rangle$, is inductively defined by the following equations.

- If t is an expression, then $\langle t \rangle$ has the type $[exp]$.

$$\begin{array}{lcl}
\langle \mathrm{VAR}(st) \rangle & = & Const(\text{“VAR”})\ \$\ Const(st) \\
\langle \mathrm{CON}(st) \rangle & = & Const(\text{“CON”})\ \$\ Const(st) \\
\langle \mathrm{ITE}(e_1, e_2, e_3) \rangle & = & ((Const(\text{“ITE”})\ \$\ \langle e_1 \rangle)\ \$\ \langle e_2 \rangle)\ \$\ \langle e_3 \rangle \\
\langle \mathrm{LET}(st, e_1, e_2) \rangle & = & (Const(\text{“LET”})\ \$\ Abs(st, \langle e_2 \rangle))\ \$\ \langle e_1 \rangle \\
\langle \mathrm{AP}(e_1, e_2) \rangle & = & (Const(\text{“AP”})\ \$\ \langle e_1 \rangle)\ \$\ \langle e_2 \rangle \\
\langle \mathrm{LAM}(st, e) \rangle & = & Const(\text{“LAM”})\ \$\ Abs(st, \langle e \rangle)
\end{array}$$

- If t is a *pat*, a *decl*, or a *prog*, then

$$\begin{array}{lcl}
\langle \texttt{nil} \rangle & = & Const(\text{“}nil\text{”}) \\
\langle p :: ps \rangle & = & (Const(\text{“}::\text{”})\ \$\ \langle p \rangle)\ \$\ \langle ps \rangle \\
\\
\langle \mathrm{DECL}(p, e) \rangle & = & (Const(\text{“DECL”})\ \$\ \langle p \rangle)\ \$\ \langle e \rangle \\
\\
\langle \texttt{nil} \rangle & = & Const(\text{“}nil\text{”}) \\
\langle d :: ds \rangle & = & (Const(\text{“}::\text{”})\ \$\ \langle d \rangle)\ \$\ \langle ds \rangle
\end{array}$$

  Note the difference between the ML constructors `nil` and `::` on the left-hand sides and the meta constructors *nil* and :: on the right-hand sides.

The definition of $\langle t \rangle$ uses the parameterized type $[list\ '\alpha]$ to describe the different syntactic sorts: *pat* and *prog*.

As an example, the expression LET $u = v$ IN LET $v = 5$ IN $u * v$ is the following term in the object-syntax:

$$\begin{array}{l}
\mathrm{LET}(\ \text{“}u\text{”}, \mathrm{VAR}(\text{“}v\text{”}), \\
\quad \mathrm{LET}(\text{“}v\text{”}, \mathrm{CON}(\text{“}5\text{”}), \\
\quad\quad \mathrm{AP}(\mathrm{AP}(\mathrm{CON}(\text{“}*\text{”}), \mathrm{VAR}(\text{“}u\text{”})), \mathrm{VAR}(\text{“}v\text{”})))).
\end{array}$$

The above object-term is more complicated than the expression itself. It can again be presented by the following meta-term of the type $[exp]$, which is even more complicated than the object-term.

$$
\begin{array}{l}
(Const(\text{“}\textsc{let}\text{”})\ \$ \\
\quad Abs(\text{“}u\text{”}, \\
\quad\quad (Const(\text{“}\textsc{let}\text{”})\ \$ \\
\quad\quad\quad Abs(\text{“}v\text{”}, \\
\quad\quad\quad\quad (Const(\text{“}\textsc{ap}\text{”})\ \$ \\
\quad\quad\quad\quad\quad ((Const(\text{“}\textsc{ap}\text{”})\ \$ \\
\quad\quad\quad\quad\quad\quad (Const(\text{“}\textsc{con}\text{”})\ \$\ Const(\text{“}*\text{”})))\ \$ \\
\quad\quad\quad\quad\quad\quad (Const(\text{“}\textsc{var}\text{”})\ \$\ Const(\text{“}u\text{”}))))\ \$ \\
\quad\quad\quad\quad\quad (Const(\text{“}\textsc{var}\text{”})\ \$\ Const(\text{“}v\text{”}))))\ \$ \\
\quad\quad\quad (Const(\text{“}\textsc{con}\text{”})\ \$\ Const(\text{“}5\text{”}))))\ \$ \\
\quad\quad (Const(\text{“}\textsc{var}\text{”})\ \$\ Const(\text{“}v\text{”}))
\end{array}
$$

The complication of internal forms is no problem in an actual (transformation) system, if we have a way to present terms in a suitable external form and to transform the external form to the internal form.

5.2 Representation of Program Transformations

We have seen that using meta-syntax to express syntactical objects of an object language is very complicated, and the representations are usually difficult to understand. To make the abstract representations more readable, some abbreviations will be introduced.

Convention:

- $t_1\ t_2 \cdots t_n$ denotes $(\cdots(t_1\ \$\ t_2)\ \$\ \cdots)\ \$\ t_n$.
- $\lambda st.t$ denotes $Abs(st, t)$.
- X denotes $Free(\text{“}X\text{”})$.
- Lists at the meta-level will be written as comma-separated components between brackets, e.g., $[a, b]$ denotes $(Const(\text{“}::\text{”})\ \$\ a)\ \$\ ((Const(\text{“}::\text{”})\ \$\ b)\ \$\ Const(\text{“}nil\text{”}))$.
- A constant of the form $Const(\text{“}c\text{”})$ that denotes a constructor of the object-syntax will be denoted by $\langle c\rangle$, such as $\langle\text{LET}\rangle$, $\langle\text{ITE}\rangle$, and so on.
- Variables or constants of the object language are usually represented as $Const(\text{“VAR”})\ \$\ Const(\text{“}x\text{”})$ and $Const(\text{“CON”})\ \$\ Const(\text{“}a\text{”})$ in the meta-syntax respectively. For simplicity, we will abbreviate them as $\langle\text{“}x\text{”}\rangle$ and $\langle\text{“}a\text{”}\rangle$, so the object-expression $u * v$ will be written as $\langle\text{“}u\text{”}\rangle\ \langle\text{“}*\text{”}\rangle\ \langle\text{“}v\text{”}\rangle$.

In fact, variables and constants have no essential differences at the meta-level. However, we distinguish them to obtain more information about the object level through combinators, which will be seen later.

5.2.1 Review of Motivating Examples

Example 1

Under these conventions, the presentation of the program transformation rule LET $Y = A$ IN $B \implies B[A/Y]$ is now

$$\langle\text{LET}\rangle\ (\lambda Y.B)\ A \implies (\lambda Y.B)\ A, \qquad (1)$$

where Y, B, A are all free meta-variables.

The expression LET $u = v$ IN LET $v = 5$ IN $u * v$ will be represented as

$$\langle\text{LET}\rangle\ (\lambda\text{“}u\text{”}.\ \langle\text{LET}\rangle\ (\lambda\text{“}v\text{”}.Bound(1)\ \langle\text{“}*\text{”}\rangle\ Bound(0))\ \langle\text{“}5\text{”}\rangle)\ \langle\text{“}v\text{”}\rangle\,.$$

The left-hand side of rule (1) matches the above term with the substitution:

$$\{Y \mapsto \text{“}u\text{”}, A \mapsto \langle\text{“}v\text{”}\rangle\,, B \mapsto \langle\text{LET}\rangle\ (\lambda\text{“}v\text{”}.Bound(1)\ \langle\text{“}*\text{”}\rangle\ Bound(0))\ \langle\text{“}5\text{”}\rangle\}.$$

The term can be transformed to

$$\begin{array}{ll} & (\lambda\text{“}u\text{”}.\ \langle\text{LET}\rangle\ (\lambda\text{“}v\text{”}.Bound(1)\ \langle\text{“}*\text{”}\rangle\ Bound(0))\ \langle\text{“}5\text{”}\rangle)\ \langle\text{“}v\text{”}\rangle \\ \to_\beta & \langle\text{“LET”}\rangle\ (\lambda\text{“}v\text{”}.\ \langle\text{“}v\text{”}\rangle\ \langle\text{“}*\text{”}\rangle\ Bound(0))\ \langle\text{“}5\text{”}\rangle\,. \end{array}$$

Again, the left-hand side of the transformation rule matches the new term with the substitution:

$$\{Y \mapsto \text{“}v\text{”}, A \mapsto \langle\text{“}5\text{”}\rangle\,, B \mapsto \langle\text{“}v\text{”}\rangle\,\langle\text{“}*\text{”}\rangle\,Bound(0)\}.$$

The term is further transformed to

$$\begin{array}{ll} & (\lambda\text{“}v\text{”}.\,\langle\text{“}v\text{”}\rangle\,\langle\text{“}*\text{”}\rangle\,Bound(0))\ \ \langle\text{“}5\text{”}\rangle \\ \to_\beta & \langle\text{“}v\text{”}\rangle\,\langle\text{“}*\text{”}\rangle\,\langle\text{“}5\text{”}\rangle\,. \end{array}$$

So, the problem of "variable capturing" stated in our first motivating example is solved by the β-reduction of the higher-order abstract syntax.

Example 2

The following transformation rule

$$C\ (\text{IF}\ P\ \text{THEN}\ S_1\ \text{ELSE}\ S_2)\ \Longrightarrow\ \text{IF}\ P\ \text{THEN}\ C\ S_1\ \text{ELSE}\ C\ S_2$$

can be represented as

$$C\ (\langle\text{ITE}\rangle\ \ P\ S_1\ S_2) \Longrightarrow\ \langle\text{ITE}\rangle\ \ P\ (C\ S_1)\ (C\ S_2) \qquad (2)$$

The expression to be transformed is now in the form

$$\langle\text{LET}\rangle\ \ (\lambda\text{“}b\text{”}.\,\langle\text{ITE}\rangle\ \ Bound(0)\ \ \langle\text{“}1\text{”}\rangle\ \ \langle\text{“}2\text{”}\rangle)\ \ \langle\text{“FALSE”}\rangle\,.$$

Matching the term against the left-hand side of the transformation rule, we get two substitutions:

$$\{C \mapsto \lambda\text{“}x\text{”}.(\langle\text{LET}\rangle\ \ (\lambda\text{“}b\text{”}.\,\langle\text{ITE}\rangle\ \ Bound(0)\ \ \langle\text{“}1\text{”}\rangle\ \ \langle\text{“}2\text{”}\rangle)\ \ \langle\text{“FALSE”}\rangle\}$$

$$\begin{array}{l}\{C \mapsto \lambda\text{“}x\text{”}.\,\langle\text{LET}\rangle\ \ (\lambda\text{“}v\text{”}.Bound(0))\ \ \langle\text{“FALSE”}\rangle\,, P \mapsto Bound(0), \\ S_1 \mapsto \langle\text{“}1\text{”}\rangle\,, S_2 \mapsto \langle\text{“}2\text{”}\rangle\}\end{array}$$

The first-substitution leaves the free variables P, S_1 and S_2 not substituted. The second-substitution substitutes P with $Bound(0)$, the bound variable leaves its scope. The first-substitution will issue an uninteresting transformation; and the second one will cause the incorrect transformation shown by the second motivating example.

To exclude such kind of substitutions, we will present some useful matching combinators in the rest of this section. Using them, we can define a lot of interesting conditions for transformation rules.

5.2.2 Some Useful Combinators

An important property of extended higher-order abstract syntax is the introduction of combinators. The matching combinators or functions used in the previous chapters can be encoded in the abstract syntax. Here, we take $++$ as an example. The type T_{++} of the combinator $++$ is:

$$[list\,'\alpha\ \to\ list\,'\alpha\ \to\ list\,'\alpha]$$

The two equations eqs_{++} for ++ are

$$\begin{array}{l} Eq(\langle\text{“}nil\text{”}\rangle\ \langle\text{“}{++}\text{”}\rangle\ YS,\ YS) \\ Eq((X\ \langle\text{“}::\text{”}\rangle\ XS)\ \langle\text{“}{++}\text{”}\rangle\ YS,\ X\ \langle\text{“}::\text{”}\rangle\ (XS\ \langle\text{“}{++}\text{”}\rangle\ YS)) \end{array}$$

So, the representation of ++ is

$$Comb(\text{“}{++}\text{”}, T_{++}, eqs_{++}).$$

Two very important conditions for program transformation rules are:

- All free variables on the right-hand side of a transformation rule should be substituted during any application of the rule, otherwise a program may be transformed to a new one that contains free variables.
- In any substitution, bound variables should not leave their scope, otherwise a program will be transformed to a new one that contains bound variables without scope.

For this purpose, we define two combinators: *noFree* and *noBoundF*, with the following types:

$$\begin{array}{ll} noFree: & [exp \rightarrow bool] \\ noBoundF: & [nat \rightarrow exp \rightarrow bool] \end{array}$$

$$\begin{array}{lcl} noFree(Free(st)) & = & false \\ noFree(Const(st)) & = & true \\ noFree(Bound(n)) & = & true \\ noFree(Abs(st,t)) & = & noFree\ t \\ noFree(t_1\ \$\ t_2) & = & and\ (noFree\ t_1)\ (noFree\ t_2) \\ & & \\ noBoundF\ m\ (Bound(n)) & = & m > n \\ noBoundF\ m\ (Const(st)) & = & true \\ noBoundF\ m\ (Free(st)) & = & true \\ noBoundF\ m\ (Abs(st,t)) & = & noBoundF\ (m+1)\ t \\ noBoundF(t_1\ \$\ t_2) & = & and\ (noBoundF\ m\ t_1)\ (noBoundF\ m\ t_2) \end{array}$$

The function $>$ is defined on natural numbers. *noFree* t is *true* if there is no free variable in the term t; *noBoundF* 0 t results in *true* if there is no bound variable in t that leaves its scope. For example,

$$noBoundF\ 0\ (Abs(\text{“}x\text{”}, Bound(1)))$$

is equal to *false*. We do not give equations about terms of kind *Comb*, since no expressions of the object language will be expressed by combinators.

Now, we can associate transformation rule (2) with the following context equations:

$$\begin{array}{ll} [EQ(noFree\ P, true), & EQ(noBoundF\ 0\ P, true) \\ EQ(noFree\ S_1, true), & EQ(noFree\ S_2, true)] \end{array}$$

Under these conditions, both substitutions in example 2 are non-satisfiable matches. That is, the rule is not applicable to the term.

In appendix B, some more useful combinators are listed. Among them are:

- *eqExp* and *subExp*, which are often used to test the syntactical equivalence and the subexpression relation between object expressions.

- *freeIn* and *boundIn*, which prove whether a variable is free or bound in an expression at the object-level. They can be used to construct the condition in transformation rule (3) of the third motivating example.

There are still some useful functions that cannot be expressed as combinators. *typeOf* that evaluates the semantic type of an expression is such an example. When lifting an object equation to a transformation rule, we usually need "type-preservation". For instance, lifting the equation

$$\forall a \colon \mathrm{N{\scriptstyle AT}}.\ a + 0 \ = \ a$$

produces the following transformation rule

$$V + 0 \implies V \ \Leftarrow\ [EQ(\mathit{typeOf}\, V, \mathrm{N{\scriptstyle AT}})].$$

To solve this problem, we are required to represent types of the object language using the meta-syntax. The presentation of all expressions as terms of the type [*exp*] is not enough. If type constructors of the object language are assumed to be FUNTYPE, BOOLEAN, and NAT, then we can associate them with the following meta-types:

$$\begin{array}{lcl}
\langle \mathrm{ITE} \rangle & : & [\mathrm{B{\scriptstyle OOLEAN}} \to \ '\alpha \to \ '\alpha \to \ '\alpha] \\
\langle \mathrm{LET} \rangle & : & [('\alpha_1 \to \ '\alpha_2) \to \ '\alpha_1 \to \ '\alpha_2] \\
\langle \mathrm{AP} \rangle & : & [(\mathrm{F{\scriptstyle UNTYPE}}\, '\alpha_1\, '\alpha_2) \ \to \ '\alpha_1 \to \ '\alpha_2] \\
\langle \mathrm{LAM} \rangle & : & ['\alpha_1 \to \ '\alpha_2 \to (\mathrm{F{\scriptstyle UNTYPE}}\, '\alpha_1\, '\alpha_2)]
\end{array}$$

But, under this new representation, we lose the possibility to define general combinators on expressions like *noFree* and *noBoundF*.

5.3 Outline of a Transformation System Prototype

Most frameworks for the development of correct software are based on some notion of refinement, i.e., a correctness-preserving development step. In the methodology of program development by transformation, for example, CIP [6, 7] or PROSPECTRA [37], such a development corresponds to the application of a pre-conceived transformation rule or method. The tasks of software construction and verification are integrated, both are carried out as fundamental parts of the development process. The implementor starts with a formal requirements specification, and then gradually refines it into an optimized, executable, possibly machine-oriented specification or program. The developer has to have expertise in choosing the right development step and creativity to come up with solutions. The advantage of the transformational approach is that each transformation rule is preconceived as a problem solution or optimization schema; it is clear what the parameters and proof obligations are at each step. Elementary transformations are considered as the fundamental structures through which development goals are achieved.

Transformation can be seen as a form of deduction. Applying transformation rules in a kind of rewriting process is a stepwise development of programs,

$$T_1 \Longrightarrow T_2 \Longrightarrow \cdots \Longrightarrow T_n,$$

where each elementary transformation $T_i \Longrightarrow T_{i+1}$ $(1 \leq i < n)$ is an application of a transformation rule $P \Longrightarrow T$ to the program T_i. Because of the introduction of higher-order terms and matching combinators, the elementary transformation is in fact a relation instead of a function: a program can be transformed to some different new programs in one step.

The purpose of the transformation system prototype is to support the framework of generic transformational developments described in [48]. So, it should consist of

- the abstract representation of syntactical objects associated with programs and program transformations to support generic transformations;
- the implementation of elementary transformations through a powerful notion of extended matching;
- the realization of a transformational calculus and a tactical calculus with a number of desirable algebraic properties, which is the basis for a formalization of transformational developments; and
- a theorem prover to guarantee the semantic correctness of transformations.

Transformation calculus and tactical calculus

A transformation calculus together with a tactical calculus is introduced to compose elementary transformations to developments, and to formally represent and to reason over transformational developments. Combining forms have been known as tacticals to make formal calculi applicable for complex derivations. In transformation systems, most of the notable examples like CIP-S [7] and PROSPECTRA [37] provide rudimentary "control tactics" or "control languages". Note that tacticals provided by Isabelle and the tacticals for program transformations are different.

Since a transformation is regarded as a relation, we can compose transformations by the tactical composition operator $\circ$. Transformations can also be combined by the disjoint union operator $\sqcup$. The orelse operator $\triangleright$ means "if the first transformation is not applicable, try the second one". One advantage of a tactical calculus is the set of algebraic properties that can be used to formally manipulate tactical expressions and can be verified on the semantic basis of the calculus. The following laws are some examples from [83] (let T_1, T_2, and T_3 be any transformation rules).

$$\begin{array}{ll}(T_1 \circ T_2) \circ T_3 \equiv T_1 \circ (T_2 \circ T_3) & \\ T_1 \circ (T_2 \sqcup T_3) \equiv (T_1 \circ T_2) \sqcup (T_1 \circ T_3) & (T_1 \sqcup T_2) \circ T_3 \equiv (T_1 \circ T_3) \sqcup (T_2 \circ T_3) \\ (T_1 \triangleright T_2) \triangleright T_2 \equiv T_1 \triangleright T_2 & T_1 \triangleright (T_1 \triangleright T_2) \equiv T_1 \triangleright T_2\end{array}$$

The transformation calculus presented in [83] enriches elementary transformations with annotation constraints and higher-order transformations. The position where a transformation ought to be applied in a term can be presented in a very elegant fashion through the following higher-order transformation rule:

$$C \Longrightarrow (P \Longrightarrow T) \Longrightarrow (C\ P \Longrightarrow C\ T), \qquad \text{(lift-context)}$$

where C, P and T can be seen as terms over the extended abstract syntax. The following example may demonstrate the use of this higher-order transformation rule. We take the transformation rule

$$\langle \text{LET} \rangle\ Y\ A\ B\ \Longrightarrow\ B[A/Y] \qquad \text{(let)}$$

to transform the expression

$$\langle \text{LET} \rangle\ \langle \text{“}u\text{”} \rangle\ \langle \text{“}v\text{”} \rangle\ (\langle \text{LET} \rangle\ \langle \text{“}v\text{”} \rangle\ \langle \text{“5”} \rangle\ (\langle \text{“}u\text{”} \rangle\ \langle \text{“}*\text{”} \rangle\ \langle \text{“}v\text{”} \rangle)).$$

Note that instead of using higher-order representation, we represent a "LET" expression $\text{LET}(u, e_1, e_2)$ in the following first-order form:

$$\langle \text{LET} \rangle\ \langle \text{“}u\text{”} \rangle\ \langle e_1 \rangle\ \langle e_2 \rangle\,.$$

The higher-order form of the transformation rule obtained by applying (lift-context) to (let) is

$$\begin{array}{rl} C & \Longrightarrow\ (\langle \text{LET} \rangle\ Y\ A\ B \Longrightarrow B[A/Y]) \\ & \Longrightarrow\ (C\ (\langle \text{LET} \rangle\ Y\ A\ B) \Longrightarrow C\ (B[A/Y])). \end{array}$$

Let C be the context

$$\langle \text{LET} \rangle\ \langle \text{“}u\text{”} \rangle\ \langle \text{“}v\text{”} \rangle\ \Box.$$

That is, we apply the rule first to the inner $\langle \text{LET} \rangle$ expression. The expression is transformed to

$$\langle \text{LET} \rangle\ \langle \text{“}u\text{”} \rangle\ \langle \text{“}v\text{”} \rangle\ (\langle \text{“}u\text{”} \rangle\ \langle \text{“}*\text{”} \rangle\ \langle \text{“5”} \rangle).$$

Now let C be the context $\Box$, the expression is transformed to

$$\langle \text{“}v\text{”} \rangle \langle \text{“}*\text{”} \rangle \langle \text{“5”} \rangle\,.$$

This example also shows that the "variable capturing" problem is avoided by using the "innermost strategy".

The theorem prover: Isabelle

Isabelle is a generic theorem prover, designed for interactive reasoning in a variety of formal theories. Its logical foundations are: representing logical syntax in the typed λ-calculus; expressing inference rules in Isabelle's meta-logic; and proof construction.

Higher-order unification is central to theorem proving. It supports quantifier reasoning by allowing certain "schematic variables" to be instantiated later. Logically, schematic variables are free variables for unification. Pragmatically, they may be instantiated during a proof, while ordinary variables remain fixed. The higher-order unification in Isabelle does not support unification modulo equations.

Theories are modules that contain information on a logic, or definitions that describe the problem being modelled, in the case of applications. Theories are often defined as extensions of other theories. When using Isabelle for reasoning about some domain, one writes theories as extensions to existing logics and theories in which underlying types and data structures are made available. In a theory, one can define several things, such as new types and data structures, new constants and their associated rules.

To implement a prototyping program transformation system supported by the Isabelle system, we should define the following things using theories provided by Isabelle:

- The specifications defining matching combinators.
- The transformation calculus and the tactical calculus.
- The object logic for transformations.

Isabelle's user interface is ML. Proofs are carried out by applying certain ML functions that change a stored proof state. The extended matching algorithm we have implemented will be used as an ML function.

- Elementary transformations will be supported by powerful extended matching.
- Properties of the tactical calculus can be verified by Isabelle.
- Proof obligations for correctness-preserving transformations can also be reflected by the proofs in Isabelle.

In the rest of this chapter, we will concentrate on the implementation of the matching algorithm $\mathcal{MA}^2$.

5.4 Implementation of the Matching Algorithm

In chapter 2, 3, and 4, we have developed the matching algorithm $\mathcal{MA}^2$ in a gradual way:

$$\begin{array}{c}\text{the first-order matching algorithm } \mathcal{MA}^1\\ \downarrow\\ \text{the improvements } \mathcal{MA}^1_d,\ \mathcal{MA}^1_p, \text{ and } \mathcal{MA}^1_c\\ \downarrow\\ \text{the contextual matching algorithm } \mathcal{MA}^C\\ \downarrow\\ \text{the higher-order matching algorithm } \mathcal{MA}^2\end{array}$$

The implementation of the matching algorithm is constructed under the following three considerations:

- The parallel implementation during the development of the algorithm.
- The use of existing algorithms and techniques.
- The possibility for further improvements.

5.4.1 The Basic Structure

The two most important data structures used to implement the matching algorithm are *term* and *typ*, which are defined in the first section of this chapter and compatible with the terms and types used in the Isabelle system. The search spaces of the matching process are recorded by the datatype *matTree*, which uses the datatype *matProb* for matching problems. *matProb* is a list of matching pairs of the type *matPair*.

$$\begin{array}{lcl}\texttt{datatype}\ \mathit{matPair} & = & \mathit{mkPair}\ \texttt{of}\ \mathit{term} * \mathit{term}\\[1ex] \texttt{type}\ \mathit{matProb} & = & \mathit{matPair}\ \texttt{list}\\[1ex] \texttt{datatype}\ \mathit{matTree} & = & \mathit{mkLeaf}\ \texttt{of}\ \mathit{matProb}\\ & | & \mathit{mkTree}\ \texttt{of}\ \mathit{matProb} * (\mathit{matTree}\ \texttt{list})\end{array}$$

The basic structure of the implementation of $\mathcal{MA}^1$ is given in figure 10, in which bold arrows show the dataflow, boxes present modules, and thin arrows denote the dependency relations between modules. A module may contain several other submodules, e.g. "Matching-Tree Construction" contains "simplification", "first-order unification", and "first-order structural narrowing".

The important techniques applied for first-order extended matching are substitution, first-order term rewriting, and first-order unification. All of these are well studied and widely used.

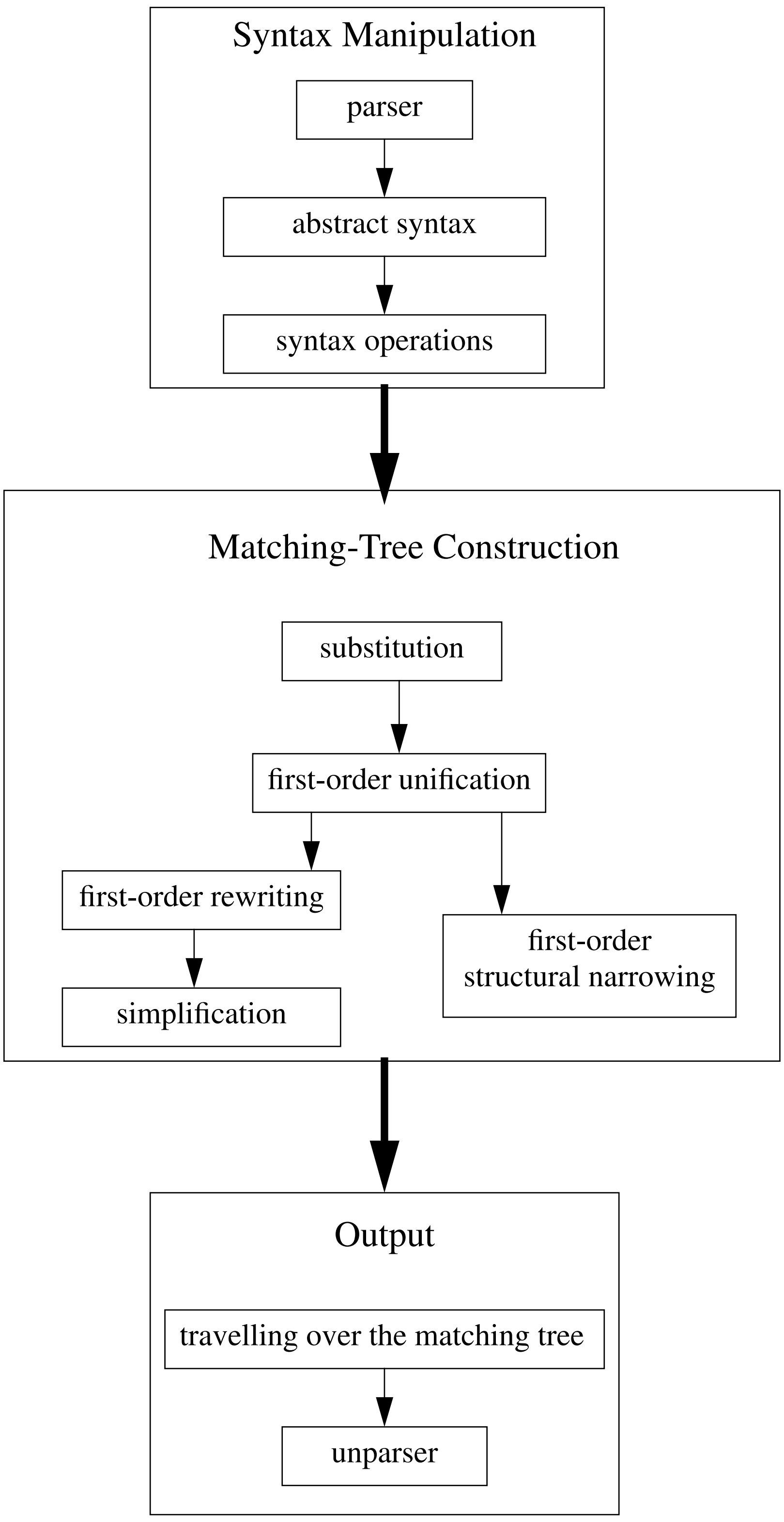

Figure 10: The basic module structure of $\mathcal{MA}^1$

5.4.2 Improvements and Contextual Matching

In section 2.3, we have shown that the first version of the implementation is rather inefficient. To improve the efficiency, three techniques are applied: dependency analysis, partial sharing analysis, and commutative matching. The extension with contextual matching is direct and easy, but it makes the implementation much more interesting. The improved structure is shown in figure 11.

To add dependency analysis into the implementation, we need to change the module "structural narrowing" to the new module "solution", which contains a new function "choosing" to choose a solvable matching pair. All other things remain unchanged.

To implement partial sharing analysis, we have introduced the datatype *sPair* and the type *sList*.

$$\begin{array}{lcl} \texttt{datatype}\ sPair & = & mkSP\ \texttt{of}\ term * term * (matProb\ \texttt{list}) \\ \\ \texttt{type}\ sList & = & sPair\ \texttt{list} \end{array}$$

A new function "look_up" is added to find solutions to a solvable matching pair, if its similar pair is already solved.

For commutative matching, we have first extended the "rewriting" module to "C-rewriting" that realizes both normal term rewriting and rewriting modulo commutativity. We will not discuss commutative matching here, since it is presently only implemented for a small set of functions like $+$, $*$, max.

The extension to contextual matching is quite easy. Two of three basic modules to compute a context are already there: "substitution" and "rewriting". The new module "context checker" is introduced to test whether two terms are constant equivalent or non-equivalent. See section 3.1 for more details.

5.4.3 Extension to the Higher-Order Case

An interesting and also the most difficult extension is to consider extended higher-order matching.

Name-free representation and λ-reductions

Internal terms are expressed using name-free representation. To ease implementation of the parser and unparser, we have introduced a datatype *exp*, which is very similar to the datatype *term*, except that it does not use name-free representations. Two ML functions are defined to transform terms between these two presentations:

$$\begin{array}{ll} toTerm: & exp \rightarrow term \\ toExp: & term \rightarrow exp. \end{array}$$

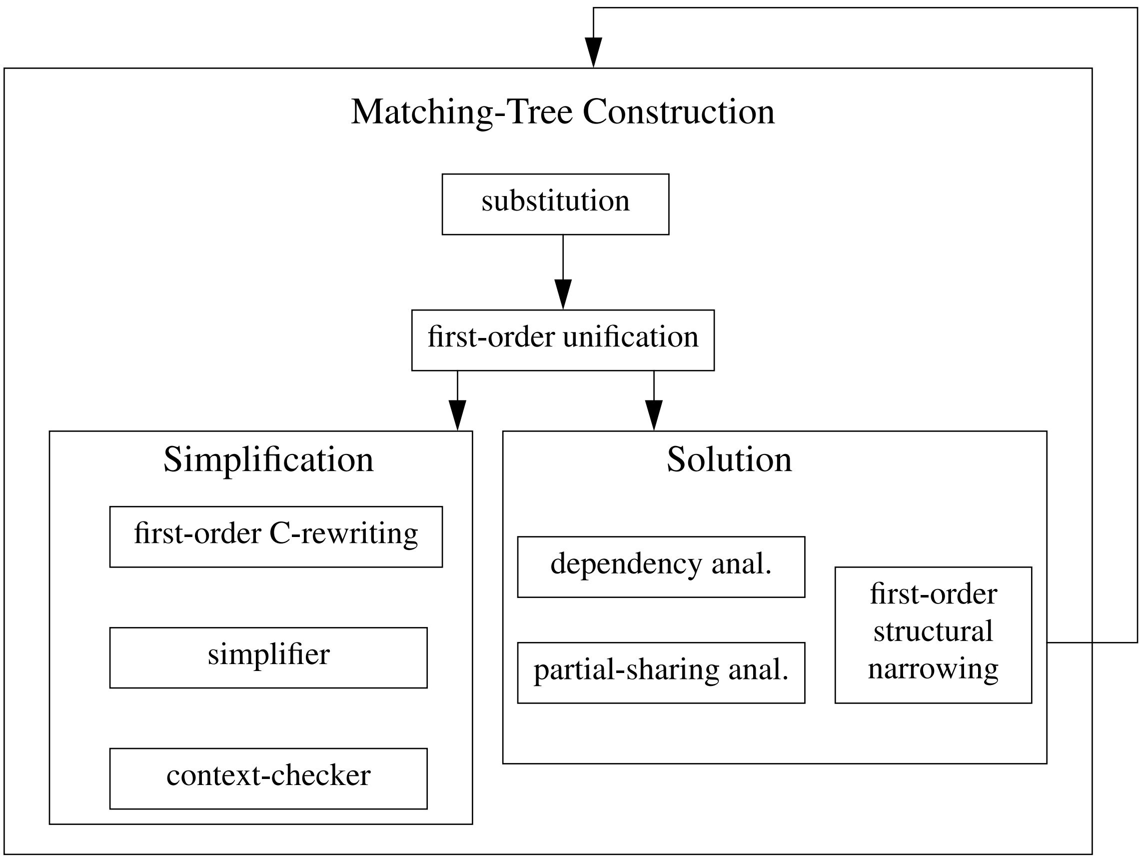

Figure 11: Some improvements

To treat higher-order terms, of course, we need the β-reduction and the η-reduction. Terms are always kept in long $\beta\eta$-normal form. Three often used functions are:

- *betaNF* to reduce a term to its β-normal form;
- *etaNF* to reduce a term to its η-normal form; and
- *etaLNF* to transform a β-normal form to its long $\beta\eta$-normal form.

Higher-order pattern unification

A unification algorithm for higher-order patterns was first put forward by Miller in 1991 [55]. Nipkow presented a functional version of the algorithm in [60]. In appendix A, we present the algorithm from [60]. The pattern unification algorithm implemented by Nipkow for the untyped case has been extended here to the simply typed case.

With the help of pattern unification, we can now describe higher-order rewriting and higher-order structural narrowing. We have first implemented a "higher-order renaming with respect to a set of variables". In the first order case, to rename an equation is quite easy: introducing some new variables to rename the variables in the equation, their types stay unchanged. In the higher-order case, we have to change the names and the types as well.

To narrow (or to rewrite) a subterm of a higher-order term t at position p with an equation $l = r$, we should consider all bound variables of the subterm. Let $\{x_1, \cdots, x_k\}$ be the set of bound variables from the root of t to the position p, we first $\{x_1, \cdots, x_k\}$-lift l:

$$\sigma = \{F \mapsto (\rho F)(x_1, \cdots, x_k) \mid F \in \mathcal{FV}(l)\},$$

where ρ is a renaming such that $\mathcal{D}(\rho) = \mathcal{FV}(l)$, $\rho F : \tau_1 \to \cdots \to \tau_k \to \tau$ if $x_i : \tau_i$ $(1 \leq i \leq k)$, and $F : \tau$. The names of free variables in l are changed and their types are lifted. The subterm will then be narrowed (or rewritten) to the instance of r.

The rest is similar to the implementation of the first-order rewriting and structural narrowing, because the left-hand side of each equation is a pattern, the term to be rewritten or to be narrowed is also a pattern, so the most general match (in the case of rewriting) or the most general unifier (in the case of narrowing) exist uniquely.

Second-order matching

In algorithm $\mathcal{MA}^2$ we made an important assumption: For the class of pure λ-terms concerned, there exists a matching algorithm that always produces a finite complete set of minimal matches for this class of higher-order matching problems. In practice, we have only considered the second-order case and have taken Huet and Lang's matching algorithm [40], which always produces a finite complete set of minimal matches for second-order matching problems. A term t is said to be **second-order**, if the types of all its variables are either first-order or second-order, and the types of all its constants are no more than third-order. The second-order case covers most applications in program transformations, and the matching algorithm is more efficient and easier to implement than some other higher-order matching algorithms like the third-order one in [18]. We have implemented the second-order matching algorithm from Huet and Lang. A description of the second-order matching algorithm can be found in appendix A.

Combining all this, we have a new structure of the implementation, see figure 12.

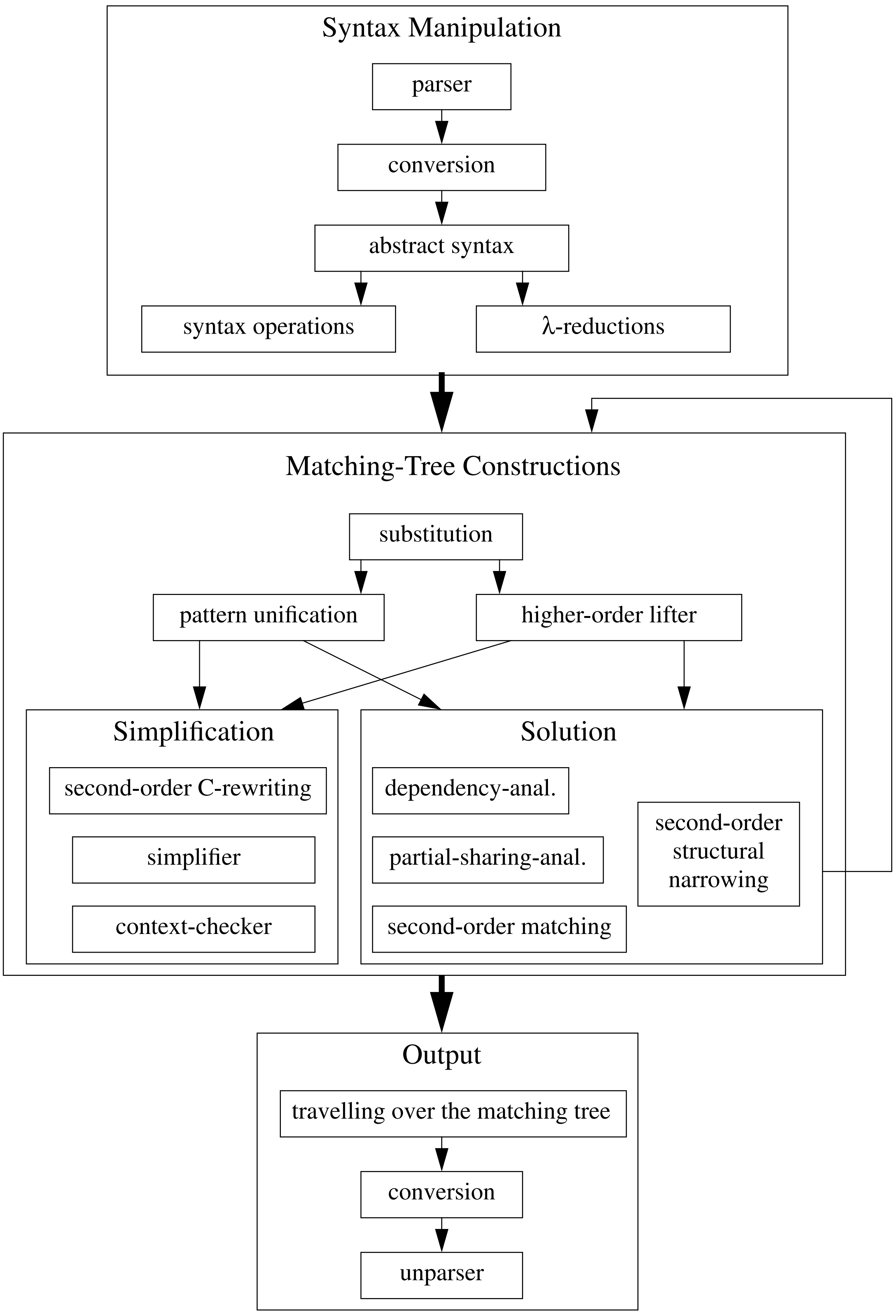

Figure 12: The module structure of $\mathcal{MA}^2$

Chapter 6

Conclusions and Future Researches

In this thesis, extended matching of both first-order and higher-order has been presented and studied as a promising support for program manipulation and other formal systems where matching is a central operation. Particularly, its application in program transformation systems has been discussed.

6.1 Summary of Major Results

An important feature of the work is that we have paid attention to both theoretical and practical aspects, which can be seen from the following points:

- the matching algorithms have desirable properties, such as completeness and termination, and their implementations are easy and efficient;
- a special case of $\mathcal{E}$-matching has been considered in this thesis, and a large number of examples have demonstrated its powerfulness in practical applications;
- contextual matching as a supplement of the theoretical framework has provided an important practical extension.

In order to solve the special class of $\mathcal{E}$-matching problems with respect to constructor-based specifications, a new equation solving technique — *structural narrowing* — has been introduced, which is easier to implement than some existing narrowing strategies, such as basic narrowing [41], normal narrowing [21] and innermost narrowing [22], and has been proved to be a complete lazy narrowing strategy unlike outermost narrowing [19]. The higher-order lifting of structural narrowing sheds some light on the possibility of efficient higher-order equation solving through narrowing in practice.

Concerning the decidable extended matching, the most important concept "non-decreasing functions" introduced in [17] has been extended in this thesis to "*proper non-decreasing functions*" in the following two aspects:

- it characterizes not only first-order functions as in [17], but also higher-order functions;
- it is appropriate for functions defined by variable-preserving equations as in [17], and by variable-dropping equations as well.

In order to make higher-order constructor-based specifications practically useful, the class of higher-order patterns discovered by Miller in [55] has been extended to *extended patterns*, which contain more useful terms and can be transformed to higher-order patterns easily, so that extended higher-order matching is efficiently solvable.

The introduction of *extended higher-order abstract-syntax* provides a generic framework for different applications of the research. As an enrichment of the higher-order abstract-syntax introduced in [66], it supports both higher-order encoding of object languages and explicit definition of constraints on syntactic objects, such that the approach also supports context-sensitive program manipulations.

6.2 Future Researches

There are some generalizations and future research topics which we feel interesting and will be summarized as follows.

Concerning the matching itself, we did not consider the relationship of extended matching and its generalization to extended unification. In fact, the important technique applied to solve equations — *structural narrowing* — can also be applied to solve unification problems with respect to constructor-based specifications. But to ensure the termination of extended unification, we should strengthen the conditions for acyclic matching combinators. The generalization will promote applications in the areas of theorem proving and functional logic programming.

Concerning the class of constructor-based specifications, we have restricted them to convergent term rewriting systems. Possible extensions of constructor-based specifications have not been discussed, nor the possible generalizations of the approach. On the one hand, we will study extended matching for a more general class of specifications, which will lead to a more general concept of extended matching. On the other hand, we should consider some important algebraic properties of functions in specifications, for instance, associativity, commutativity, and idempotence, which will cause problems of *extended matching modulo a congruence*.

Although our research should be generalized in some aspects, we will further concentrate on the applications of extended matching and efficient implementations, particularly the following research topics.

Functional programming

The interest in the theory and applications of rewriting has been growing rapidly. Narrowing, a sound and complete unification method for convergent term rewriting systems, enhances the research in the amalgamation of functional and logic programming languages.

In the functional programming world, patterns are an important concept to define functions, and pattern-matching is a crucial mechanism to evaluate functions, e.g. in the functional language ML. Although the theoretical progress in this thesis has no immediate influence, it is a prerequisite for a new concept of functional programming, as shown in the following simple examples:

$$\begin{array}{rcll} last(xs +\!\!+ [y]) & = & y & (1) \\ del(x, xs +\!\!+ [x] +\!\!+ ys) & = & xs +\!\!+ ys & (2) \end{array}$$

In both equations, the concatenation function $++$ on lists is used to describe the patterns on their left-hand sides. In (2) we also use a non-linear pattern, where the variable x occurs twice. The "functions" *last* and *del* are now defined in an elegant and abstract form with these extended patterns. Furthermore, the combination with contextual matching and higher-order matching will provide a powerful notion of pattern-matching for such extended functional programming. Of course, the following problems should be considered:

- The result of a "function call" is not necessarily unique, such as $del(0, [1, 0, 2, 0])$.
- Some additional unpredictable matching error may occur, such as $last(nil)$.

It is well-known that λ-calculus is the semantic foundation of functional languages; some interesting research in extended λ-calculus, for example, λ-calculus with patterns [61], will certainly promote the research.

Applications

Another reason to hinder the application of a powerful notion of matching in functional programming is the fact that there is still no implementation efficient enought to support this attempt. We feel that theoretical results are important, but an efficient implementation has at least the same importance. In this thesis, we have given some examples to show an obvious improvement of the matching algorithm through applying three techniques. Contextual matching also offers the possibility to reduce the matches of matching problems and speed up the matching process consequently. For practical uses, we have to develop a much more efficient matching algorithm, perhaps under some stronger restrictions. It seems that this requires a deeper understanding of application areas. For instance, to promote its application in functional languages, we need to have more knowledge about the principles of compilers for functional languages.

With respect to applications, further practical experience in program transformations, compilers, term rewriting systems, and logic systems is needed to examine whether the notion of matching studied in this thesis is adequate and strong enough in reality. A good direction would be to use the extended abstract syntax described in chapter 5 to formalize matching problems in application areas. Such a practical application may be done in a proof development system like Isabelle [64] together with the matching algorithm developed in this thesis.

Appendix A

Two Algorithms

Higher-order pattern unification

We represent here the higher-order pattern unification algorithm from Nipkow, see [60]. The algorithm is given by the following five transformation rules on unification problems. It begins with a unification problem Γ, where all terms are patterns; and terminates with Γ' in solved form, if Γ is unifiable, in which case Γ' forms the most general unifier of Γ.

(1) propagates solutions.

$$\{(\lambda x_1, \cdots, x_k.F\ x_1 \cdots x_k,\ t)\} \cup \Gamma \implies \{(F, t)\} \cup (\sigma\Gamma),$$

if $F \notin \mathcal{FV}(t)$, where $\sigma = \{F \mapsto t\}$.

(2) breaks a unification pair down into simpler ones.

$$\begin{array}{ll} & \{(\lambda x_1, \cdots, x_k.c\ s_1 \cdots s_n,\ \lambda x_1, \cdots, x_k.c\ t_1, \cdots t_n)\} \cup \Gamma \\ \implies & \{(\lambda x_1, \cdots, x_k.s_1,\ \lambda x_1, \cdots, x_k.t_1), \cdots, (\lambda x_1, \cdots, x_k.s_n,\ \lambda x_1, \cdots, x_k.t_n)\} \cup \Gamma, \end{array}$$

if c is a constructor, a function, or a bound variable.

(3) finds a partial binding for a head variable.

$$\begin{array}{ll} & \{(\lambda x_1, \cdots, x_k.F\ y_1 \cdots y_n,\ \lambda x_1, \cdots, x_k.c\ t_1, \cdots t_m)\} \cup \Gamma \\ \implies & \{(F,\ \lambda y_1, \cdots, y_n.c\ s_1 \cdots s_m)\} \cup \{(\lambda x_1, \cdots, x_k.s_1,\ \lambda x_1, \cdots, x_k.t_1), \\ & \cdots, (\lambda x_1, \cdots, x_k.s_m,\ \lambda x_1, \cdots, x_k.t_m)\} \cup (\sigma\Gamma), \end{array}$$

if $F \notin \mathcal{FV}(c\ t_1 \cdots t_m)$ and c is a constructor, a function, or a bound variable. Where $s_i = H_i\ y_1 \cdots y_n$ with H_i a new free variable $(1 \le i \le m)$, and $\sigma = \{F \mapsto \lambda y_1, \cdots, y_n.\ c\ s_1 \cdots s_m\}$.

(4) finds a unifier of a unification pair with distinct heads.

$$\begin{array}{ll} & \{(\lambda x_1, \cdots, x_k.F\ y_1 \cdots y_n,\ \lambda x_1, \cdots, x_k.G\ z_1, \cdots z_m)\} \cup \Gamma \\ \implies & \{(F,\ \lambda y_1, \cdots, y_n.H\ v_1 \cdots v_p), (G,\ \lambda z_1, \cdots, z_m.H\ v_1 \cdots v_p)\} \cup (\sigma\Gamma), \end{array}$$

if F and G are distinct free variables. Where $\{v_1, \cdots, v_p\} = \{y_1, \cdots, y_n\} \cap \{z_1, \cdots, z_m\}$, H is a new free variable, and

$$\sigma = \{F \mapsto \lambda y_1, \cdots, y_n.H\ v_1 \cdots v_p,\ G \mapsto \lambda z_1, \cdots, z_m.H\ v_1 \cdots v_p\}.$$

(5) finds a unifier of a unification pair with same head.

$$\begin{array}{ll} & \{(\lambda x_1, \cdots, x_k.F\ y_1 \cdots y_n, \lambda x_1, \cdots, x_k.F\ z_1, \cdots z_n)\} \cup \Gamma \\ \Longrightarrow & \{(F,\ \lambda y_1, \cdots, y_n.H\ v_1 \cdots v_p)\} \cup (\sigma\Gamma), \end{array}$$

where $\sigma = \{F \mapsto \lambda y_1, \cdots, y_n.H\ v_1 \cdots v_p\}$, $\{v_1, \cdots, v_p\} = \{y_i \mid y_i = z_i, 1 \leq i \leq n\}$, and H is a new free variable.

Second-order matching

Here we describe the second-order matching algorithm from Huet and Lang, see [40], by giving the following three transformation rules. It begins with a ground matching problem Γ and terminates with a matching problem Γ' in solved form, if Γ is solvable, in which case Γ' forms a minimal match of Γ.

(1) breaks a matching pair down into simpler ones.

$$\begin{array}{l} \{(\lambda x_1, \cdots, x_k.c\ s_1 \cdots s_n,\ \lambda x_1, \cdots, x_k.c\ t_1, \cdots t_n)\} \cup \Gamma \\ \Longrightarrow \{(\lambda x_1, \cdots, x_k.s_1,\ \lambda x_1, \cdots, x_k.t_1), \cdots, (\lambda x_1, \cdots, x_k.s_n,\ \lambda x_1, \cdots, x_k.t_n)\} \cup \Gamma, \end{array}$$

if c is a constructor, or a bound variable.

(2) finds a partial binding for a head variable.

$$\begin{array}{l} \{(\lambda x_1, \cdots, x_k.F\ t_1 \cdots t_n,\ \lambda x_1, \cdots, x_k.c\ s_1, \cdots s_m)\} \cup \Gamma \\ \Longrightarrow \{(F,\ \lambda y_1, \cdots, y_n.c\ t'_1 \cdots t'_m), (\lambda x_1, \cdots, x_k.t'_1,\ \lambda x_1, \cdots, x_k.s_1), \cdots, \\ \qquad\qquad (\lambda x_1, \cdots, x_k.t'_m,\ \lambda x_1, \cdots, x_k.s_m)\} \cup (\sigma\Gamma), \end{array}$$

if F is a free variable and c is a constructor or a bound variable. Where $\sigma = \{F \mapsto \lambda y_1, \cdots, y_n.\ (c\ t'_1 \cdots t'_m)\}$, each $t'_i = \lambda z_1, \cdots, z_{k_i}.\ (H_i\ y_1 \cdots y_n\ z_1 \cdots z_{k_i})$ if s_i has the type $\tau_1 \rightarrow \cdots \rightarrow \tau_{k_i} \rightarrow \tau$, and H_i is a new free variable, for all $1 \leq i \leq m$.

(3) finds a projection that is type compatible.

$$\begin{array}{l} \{(\lambda x_1, \cdots, x_k.F\ t_1 \cdots t_n,\ \lambda x_1, \cdots, x_k.s)\} \cup \Gamma \\ \Longrightarrow \{(F,\ \lambda y_1, \cdots, y_n.y_i), (\lambda x_1, \cdots, x_k.t_i,\ \lambda x_1, \cdots, x_k.s)\} \cup (\sigma\Gamma), \end{array}$$

if F is a free variable, the type of t_i $(1 \leq i \leq n)$ and the type of $F\ t_1 \cdots t_n$ are the same. $\sigma = \{F \mapsto \lambda y_1, \cdots, y_n.y_i\}$.

Appendix B

Useful Combinators

In this appendix, four useful combinators will be defined. The details of the extended higher-order abstract syntax, or meta-syntax, and of the object language can be found in section 5.1. In the definitions, we do not consider terms of the kind *Comb*, since no expression of the object language will be expressed by combinators.

eqExp and *subExp*

eqExp and *subExp* are combinators often used to test the syntactical relations between expressions. Both of them have the type:

$$[exp \to exp \to bool].$$

They will be defined recursively on the meta-syntax for terms of the type $[exp]$:

$$\begin{array}{lcl}
eqExp\ (Const(st_1))\ (Const(st_2)) & = & eqStr\ st_1\ st_2 \\
eqExp\ (Free(st_1))\ (Free(st_2)) & = & eqStr\ st_1\ st_2 \\
eqExp\ (Bound(n_1))\ (Bound(n_2)) & = & eqNat\ n_1\ n_2 \\
eqExp\ (Abs(st_1, b_1))\ (Abs(st_2, b_2)) & = & eqExp\ b_1\ b_2 \\
eqExp\ (e_1\ \$\ e_2)\ (e_3\ \$\ e_4) & = & and\ (eqExp\ e_1\ e_3)\ (eqExp\ e_2\ e_4)
\end{array}$$

Note that:

- The functions *eqStr* and *eqNat* are assumed to be predefined combinators for `string` and `nat`.
- In the fourth equation, we do not compare the names of bound variables, because of using de Bruijn indices.
- The definition is not complete; for all other cases, the results are just *false*.

$$\begin{array}{lcl}
subExp\ e\ (Const(st)) & = & eqExp\ e\ (Const(st)) \\
subExp\ e\ (Free(st)) & = & eqExp\ e\ (Free(st)) \\
subExp\ e\ (Bound(n)) & = & eqExp\ e\ (Bound(n)) \\
subExp\ e\ (Abs(st, b)) & = & or\ (eqExp\ e\ (Abs(st, b)))\ (subExp\ e\ b) \\
subExp\ e\ (e_1\ \$\ e_2) & = & or\ (eqExp\ e\ (e_1\ \$\ e_2))\ (or\ (subExp\ e\ e_1)\ (subExp\ e\ e_1))
\end{array}$$

Except for the third equation, all other equations are easy to understand. The third equation means, for example, the term $Abs(st, Bound(0))$ (i.e. $\lambda x.x$) is a subterm of

$$Abs(st_1, Abs(st_2, Bound(0)))$$

(i.e. $\lambda x, y.y$), but it is not a subterm of

$$Abs(st_1, Abs(st_2, Bound(1)))$$

(i.e. $\lambda x, y.x$). Note that no object expression will be presented as $Bound(n)$ with $n \geq 0$.

freeIn and *boundIn*

The other two useful combinators are

$$\begin{array}{ll} freeIn: & [exp \to exp \to bool], \\ boundIn: & [nat \to exp \to exp \to bool]. \end{array}$$

They check whether a variable is free or bound in an expression at the object-level. An object variable with name st is always expressed as the following term in the meta-syntax:

$$Const(\text{“VAR”})\ \$\ Const(st).$$

In the sequel, it will be denoted as V. According to our representation of the object syntax, "a variable to be free in an expression" is the same as the variable is a "subexpression of the expression", since bound variables are presented as indices in the meta-syntax. So,

$$freeIn\ V\ e = subExp\ V\ e\ .$$

$$boundIn\ m\ V\ (Const(st_1)) = false$$

$$boundIn\ m\ V\ (Free(st_1)) = false$$

$$boundIn\ m\ V\ (Bound(n)) = eqNat\ (m-1)\ n$$

$$boundIn\ m\ V\ (e_1\ \$\ e_2) = or\ (boundIn\ m\ V\ e_1)\ (boundIn\ m\ V\ e_2)$$

$$\begin{array}{l} boundIn\ m\ V\ (Abs(st_1, e)) = or\ (and\ (eqStr\ st\ st_1)\ (boundIn\ (succ\ m)\ V\ e) \\ \qquad\qquad\qquad\qquad\qquad\qquad (and\ not(eqStr\ st\ st_1)\ (boundIn\ m\ V\ e)) \end{array}$$

To check whether v is a bound variable of t, $boundIn\ 0\ \langle v\rangle\ \langle t\rangle$ will be evaluated, where $\langle e\rangle$ is the representation of any object expression e in the meta-syntax.

Bibliography

[1] Avenhaus, J. and Loría-Sáenz, C.: Higher-Order Conditional Rewriting and Narrowing. In *Proc. of First International Conference on Constraints in Computational Logics*, LNCS 845, 269-284, 1994.

[2] Aït-Kaci, H.: *Warren's Abstract Machine.* MIT Press, 1991.

[3] Bachmair, L.: *Canonical Equational Proofs.* Ronald V. Book (ed.), Birkhäuser, 1991.

[4] Barendregt, H.P.: *The Lambda Calculus.* Barwise, J., Kaplan, D., Keisler, H.J., Suppes, P. and Troelstra, A.S. (eds.) NORTH-HOLLAND, 1984.

[5] Bauer, F.L. and Wössner, H.: *Algorithmishe Sprache und Programmentwicklung.* Springer-Verlag, 1981.

[6] Bauer, F.L., Berghammer, R., Broy, M., Dosch, W., Gnatz, R., Geiselbrechtinger, F., Hangel, E., Hesse, W., Krieg-Brückner, B., Laut, A., Matzner, T.A., Möller, B., Nickl, F., Partsch, H., Pepper, P., Samelson, K., Wirsing, M. and Wössner, H.: *The Munich Project CIP, Vol. 1: The Wide Spectrum Language CIP-L.* LNCS 183, 1985.

[7] Bauer, F.L., Ehler, H., Horsch, B., Möller, B., Partsch, H., Paukner, O. and Pepper, P.: *The Munich Project CIP, Vol. 2: The Transformation System CIP-S.* LNCS 292, 1987.

[8] Bockmayr, A.: A Note on a Canonical Theory with Undecidable Unification and Matching Problem. In *Journal of Automated Reasoning*, Vol. 3, 379-381, 1987.

[9] Bosco, P.G., Giovannetti, E., Levi, G., Moiso, C. and Palamidessi, C.: A Complete Characterization of K-LEAF, a Logic Language with Partial Functions. In *Proc. of 1987 Symposium on Logic Programming*, 1987.

[10] Boyer, R.S. and Moore, J.S.: *A Computational Logic.* Academic Press, New York.

[11] Comon, H.: Equational Problems and Disunifications. In *Journal of Symbolic Computation*, 7(1989), 371-425.

[12] de Bruijn, N.G.: Lambda Calculus Notation with Nameless Dummies: A Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indagationes Mathematicae 34*, 381-392, 1972.

[13] Curry, H.B. and Feys, R.: *Combinatory Logic*, Vol. I, NORTH-HOLLAND, 1958.

[14] Dershowitz, N., and Manna, Z.: Proving Termination with Multiset Orderings. In *Communication of ACM* 22, 465-476, 1979.

[15] Dershowitz, N.: Termination of Rewriting. In: *Journal of Symbolic Computation*, 3, 1987.

[16] Dershowitz, N., Jouannaud, J.-P.: Rewrite Systems. In *Formal Models and Semantics, Handbook of Theoretical Computer Science, Vol.B*; van Leeuwen, I. (ed.) 243-320. The MIT press, 1990.

[17] Dershowitz, N., Mitra, S. and Sivakumar, G.: Decidable Matching for Convergent Systems. In *Proc. of 11th International Conference on Automated Deduction*, LNCS 607, 589-602, 1992.

[18] Dowek, G.: Third-Order Matching is Decidable. In *Proc. of 7th Annual IEEE Symposium On Logic in Computer Science*, 2-10, 1992.

[19] Echahed, R.: On Completeness of Narrowing Strategies. In *CAAP*'88, 89-101, LNCS 299, 1988.

[20] Ehrig. H, Mahr, B.: *Fundamentals of Algebraic Specification 1*, Brauer. W., Rozenberg, G. and Salomaa, A. (eds.). Springer-Verlag, 1985.

[21] Fay, M.: First-Order Unification in an Equational Theory. In *Proc. 4th Workshop on Automated Deduction*, 161-167, (1979).

[22] Fribourg, L.: SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. In *Proc. of Symposium on Logic Programming*, (1985), 172-184.

[23] Gallier and J. H., Snyder, W.: Complete Sets of Transformations for General E-Unification. LNCS 256 (1987), 256-273.

[24] Ganzinger, H.: A Completion Procedure for Conditional Equations. In *Proc. of Conditional Term Rewriting Systems*, LNCS 308, 62-83, 1987.

[25] Geser, A. and Hussmann, H.: Experiences with the *RAP* system – A Specification Interpreter Combining Term Rewriting and Resolution. In *Proc. of ESOP'86*, 339-359, LNCS 213, 1986.

[26] Giovannetti, E. and Moiso, C.: A Completeness Result for *E*-Unification Algorithms based on Conditional Narrowing. In *Proc. of the Workshop on Foundations of Logic and Functional Programming*, LNCS 306, 157-167, 1986.

[27] Goguen, J.A. and Meseguer, J.: EQLOG: Equality, Types and Generic Modules for Logic Programming. In DeGroot, D. and Lindstrom, G. (eds.) *Logic Programming: Functions, Relations and Equations*, Prentice-Hall, 295-363, 1986.

[28] Goldfarb, W.D.: The Undecidability of the Second-Order Unification Problem. *Theoretical Computer Science*, 13:225-230, 1981.

[29] Hannan, J. and Miller, D.: Use of Higher-Order Unification for Implementing Program Transformations. In Kowalski, R. and Bowen, K. (des.): *Proc. of the Logic Programming Conference*, MIT Press, 1988.

[30] Hanus, M.: Compiling Logic Programs with Equality. In *Proc. of the 2nd Int. Symposium on Programming Language Implementation and Logic Programming*, LNCS 456, 1990.

[31] Hanus, M.: Towards the Global Optimization of Functional Logic Programs. In *Proc. of Int. Conference on Compiler Construction*, LNCS 786, 1994.

[32] Harper, R., Honsell, F. and Plotkin, G.: A Framework for Defining Logics. In *Symposium on Logic in Computer Science*, 194-204, IEEE, 1987.

[33] Harper, R., MacQueen, D., Miller, R.: *Standard ML*. LFCS Report Series, ECS-LFCS-86-2. Dept. of Computer Science, University of Edinburgh, 1986.

[34] Heibrunner, S. and Hölldobler, S.: The Undecidability of the Unification and Matching Problem for Canonical Theories. In *Acta Informatica*, Vol 24, 1987.

[35] Herold, A.: Combination of Unification Algorithms. In *Proc. 8th International Conference on Automated Deduction*, LNCS 230, 450-469, 1986.

[36] Hindley, J. and Seldin, J.P.: *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.

[37] Hoffmann, B. and Krieg-Brückner, B. (eds.): *Program Development by Specification and Transformation, The PROSPECTRA Methodology, Language Family, and System*. LNCS 680, 1993.

[38] Hölldobler, S.: Foundations of Equational Logic Programming. In *Lecture Notes in Artificial Intelligence* 353, 1989.

[39] Huet, G.: A Unification Algorithm for Typed λ-calculus. In *Theoretical Computer Science*, 1:27-57, 1975.

[40] Huet, G. and Lang, B.: Proving and Applying Program Transformations Expressed with Second-Order Patterns. *Acta Informatica* 11, 1978.

[41] Hullot, J.-M.: Canonical Forms and Unification. In *Proc. 5th Workshop on Automated Deduction*, 318-334, 1980.

[42] Hussmann, H.: Unification in Conditional-Equational Theories. In *Proc. of the European Conference on Computer Algebra*, LNCS 204, 543-553, 1985.

[43] Jouannaud, J.-P. and Kirchner, H.: Completion of a Set of Rules Modulo a Set of Equations. In: *SIAM Journal of Compute* 15, 1155-1194, 1986.

[44] Jouannaud, J.-P. and Kirchner, C.: Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification. In *Computational Logic: Essays in Honor of Alan Robinson*, Lassez, J.-L. and Plotkin, G. (eds.), MIT Press, 257-321, 1991.

[45] Kaplan, S.: Simplifying Conditional Term Rewriting System: Unification, Termination and Confluence. In *Journal of Symbolic Computation 4(3)*, 295-334, 1987.

[46] Klop, J. W.: *Combinatory Reduction Systems*. Ph.D. Thesis, Mathematical Centre Tracts Nr. 127. Centre for Mathematics and Computer Science, Amsterdam, 1980.

[47] Klop, J. W.: Term Rewriting Systems. In Abramski, S., Gabbay, Dov. M. and Maibaum, T.S.E. (eds): *Handbook of Logic in Computer Science, Chapter 1*. Oxford Science Publications, 1992.

[48] Krieg-Brückner, B., Liu, J., Shi, H. and Wolff, B.: Towards Correct, Efficient and Reusable Transformational Developments. In Broy, M. and Jähnichen, S. (eds.): *KORSO, Correct Software by Formal Methods*, LNCS (to appear).

[49] Kuchen, H., Loogen, R., Moreno-Navarro, J.J. and Rodríguez-Artalejo, M.: Graph-based Implementation of a Functional Logic Language. In *Proc. of ESOP'90*, LNCS 432, 271-290, 1990.

[50] Lankford, D.S.: Some New Approaches to the Theory and Applications of Conditional Term Rewriting Systems. Report MTP-6, Dept. of Mathematics, Lousiana Tech. University, 1979.

[51] Loogen, R.: From Reduction Machines to Narrowing Machines. In *Proc. of TAPSOFT'91*, LNCS 493, 438-457, 1991.

[52] Martelli, A. and Montanari, U.: An Efficient Unification Algorithm. In *ACM Transactions on Programming Languages and Systems*, Vol 4, No. 2, 258-282, 1982.

[53] Martin, U. and Nipkow, T.: Unification in Boolean Rings. In *Proc. 8th International Conference on Automated Deduction*, LNCS 230, 506-513, 1986.

[54] Middeldorp, A. and Hamoen, E.: Completeness Results for Basic Narrowing. In *Proc. 3rd International Conference on Algebraic and Logic Programming*, LNCS 632, 244-258, 1992.

[55] Miller, D.: A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. In Schroeder-Heister, P. (ed.): *Extensions of Logic Programming*, LNCS 475, 253-281, 1991.

[56] Nadathur, G. and Miller, D.: An Overview of λProlog. In Kowalski, R.A. and Bowen, K.A. (eds.): *Proc. of 5th International Logic Programming Conference*, 810-827, 1991.

[57] Nipkow, T.: Combining Matching Algorithms: The Regular Case. In Dershowitz, N. (ed.): *Proc. 3rd Int. Conf. on Rewriting Techniques and Applications*, 343-358, LNCS 355, 1989.

[58] Nipkow, T.: Higher-Order Critical Pairs. In *Proc. of Sixth Annual IEEE Symposium on Logic in Computer Science*, 342-349, 1991.

[59] Nipkow, T.: Orthogonal Higher-Order Rewrite Systems are Confluent. In M.A. Bezem and Jan Friso Groote (eds.): *Proc. of Int. Conf. Typed Lambda Calculi and Applications*, LNCS 664, 306-317, 1993.

[60] Nipkow, T.: Functional Unification of Higher-Order Patterns. In *Proc. of the 8th IEEE Symposium of Logic in Computer Science*, 1993.

[61] van Oostrom, V.: Lambda-Calculus with Patterns. Technical Report, Dept. of Mathematics and Computer Science, Vrije University, 1990.

[62] Padawitz, P.: Strategy-Controlled Reduction and Narrowing. In *Proc. of the Conference on Rewriting Techniques and Applications*, 242-255, LNCS 256, 1987.

[63] Paulson, L.C.: The foundation of a generic theorem prover. In *Journal of Automated Reasoning*, 5:363-379, 1989.

[64] Paulson, L.C.: *ML for the Working Programmer*. Cambridge University Press, 1991.

[65] Peterson, G. E. and Stickel, M. E.: Complete Sets of Reductions for some Equational Theories. In *Journal of ACM* 28, 233-264, 1981.

[66] Pfenning, F. and Elliott, C.: Higher-Order Abstract Syntax. In *Proc. of the SIGPLAN'88, Conference on Programming Language Design and Implementation*, 1988.

[67] Pfenning, F.: Logic Programming in the LF Logical Framework. In Huet, G. and Plotkin, G. (eds.): *Logical Frameworks*, 66-78, Cambridge University Press, 1991.

[68] Plotkin, G.: Building-in Equational Theories. *Machine Intelligence,* 7, 73-90, 1972.

[69] van de Pol, J.C.: Proving Termination of Higher-Order Rewrite Systems. Technical Report 93, Department of Philosophy, Utrecht University, June, 1993.

[70] Prehofer, C.: Solving Higher-Order Equations. In *Proc. of 9th IEEE Symposium on Logic in Computer Science*, 1994.

[71] Qian, Z.: Higher-Order Equational Logic Programming. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, 1994.

[72] Remy, J.L.: *Etudes des systèms réécriture conditionnels et applications aux Types abstraits algébriques.* Thèse d'Etat, University de Nancy I, Nancy, France, 1982.

[73] Réty, R., Kirchner, C., Kirchner, H. and Lescanne, P.: NARROWER: A New Algorithm for Unification and its Application to Logic Programming. In *First Conference on Rewriting Techniques and Applications,* 141-157, LNCS 202, 1985.

[74] Schmidt-Schauss, M. : Unification in a Combination of Arbitrary Disjoint Equational Theories. *Journal of Symbolic Computation*, 8, 51-99, 1989.

[75] Shi, H.: An Efficient Matching Algorithm for Convergent Rewrite Systems. In Krieg-Brückner, B. (ed.): *Programmentwicklung durch Spezifikation und Transformation; Band II.* Informatik Bericht Nr. 10/94, Universität Bremen, 1994.

[76] Shi, H. and Wolff, B. : A Finitary Matching Algorithm for Constructor Based Theories. In Krieg-Brückner, B. (ed.): *Programmentwicklung durch Spezifikation und Transformation; Band I.* Informatik Bericht Nr. 1/94, Universität Bremen, 1994.

[77] Siekmann, J.H.: Unification Theory. *Journal of Symbolic Computation*, 7, 207-274, 1989.

[78] Snyder, W.: *Complete Sets of Transformations for General Unification.* Ph.D. Thesis, Dept. of Computer and Information Science, University of Pennsylvania, 1988.

[79] Snyder, W. and Gallier, J.: Higher-Order Unification Revisited: Complete Set of Transformations. In *Journal of Symbolic Computation,* 8, 101-140, 1989.

[80] Snyder, W. and Gallier, J.: Higher-Order E-Unification. In *Proceedings of 10th International Conference on Automated Deduction*, LNCS 449, 573-587, 1990.

[81] Stickel, M.E.: A Unification Algorithm for Associative Commutative Functions. *Journal of the ACM, Vol.28, No. 3*, 423-434, 1981.

[82] Toyama, Y., Klop, J.W. and Barendregt, H.P.: Termination for the Direct Sums of Left-Linear Term Rewriting Systems. In *Proc. of 3rd Intern. Conf. on Rewriting Techniques and Applications,* LNCS 355, 477-491, 1989.

[83] Wolff, B. and Shi, H.: Semantic Foundation of TL – A Meta-Language for Transformation. In Krieg-Brückner, B. (ed.): *Programmentwicklung durch Spezifikation und Transformation; Band I.* Informatik Bericht Nr. 1/94, Universität Bremen, 1994.

[84] Wolfram, D.A.: *The Clausal Theory of Types.* Cambridge University Press, 1994.

[85] Yamamoto, A.: Completeness of Extended Unification Based on Basic Narrowing. In *Proc. of the 7th Logic Programming Conference,* 1-10, 1989.

[86] You, Y.H.: Enumerating Outer Narrowing Derivations for Constructor Based Term Rewriting Systems. In *Journal of Symbolic Computation* 7, 391-343, 1989.

[87] Zhang, H.: Implementing Contextual Rewriting. 363-377, LNCS 656, 1993.

[88] Zhang, H. and Kapur, D.: First-Order Theorem Proving Using Conditional Rewrite Rules. In *Proc. of 9th International Conference on Automated Deduction*, LNCS 310, 1987.

[89] Zhang, H. and Remy, J.L.: Contextual Rewriting. In *Proc. of Rewriting Techniques and Application.* Dijon, France, 1985.

Index