12-2010

A TYPE ANALYSIS OF REWRITE STRATEGIES

Azamat Mametjanov
University of Nebraska at Omaha

A TYPE ANALYSIS OF REWRITE STRATEGIES

by

Azamat Mametjanov

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Doctor of Philosophy

Major: Information Technology

Under the Supervision of Dr. Victor Winter

Omaha, Nebraska

December 2010

Supervisory Committee:

Dr. Victor Winter

Dr. Mahadevan Subramaniam

Dr. Hai-Feng Guo

Dr. Ralf Lämmel

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

ABSTRACT

A TYPE ANALYSIS OF REWRITE STRATEGIES

Azamat Mametjanov

University of Nebraska at Omaha, 2010

Advisor: Dr. Victor Winter

Rewrite strategies provide an algorithmic rewriting of terms using strategic compositions of rewrite rules. Due to the programmability of rewrites, errors are often made due to incorrect compositions of rewrites or incorrect application of rewrites to a term within a strategic rewriting program. In practical applications of strategic rewriting, testing and debugging becomes substantially time-intensive for large programs applied to large inputs derived from large term grammars. In essence, determining which rewrite in what position in a term did or did not fire comes down to logging, tracing and/or diff-like comparison of inputs to outputs. In this thesis, we explore type-enabled analysis of strategic rewriting programs to detect errors statically. In particular, we introduce high-precision types to closely approximate the dynamic behavior of rewriting. We also use union types to track sets of types due to presence of strategic compositions. In this framework of high-precision strategic typing, we develop and implement an expressive type system for a representative strategic rewriting language *TL*. The results of this research are sufficiently broad to be adapted to other strategic rewriting languages. In particular, the type-inferencing algorithm does not require explicit type annotations for minimal impact on an existing language. Based on our experience with the implementation, the type system significantly reduces the time and effort to program correct rewrite strategies while performing the analysis on the order of thousands of source lines of code per second.

Category:

D.3.1[Programming Languages]: Formal Definitions and Theory

D.3.3[Programming Languages]: Language Constructs and Features

D.2.4[Software Engineering]: Software/Program Verification

General Terms: Design, Languages, Theory

Keywords: Type analysis, Term rewriting, Strategies, Program transformation, Transformation language

Contents

List of Figures

Chapter 1

Introduction

Program transformation is a mechanized manipulation of programs in order to improve them relative to a variety of criteria such as clarity, efficiency, simplicity, functionality, translation and computation. Classic program transformation is an equational reasoning framework of replacing equals by equals in all contexts of a program [12]. In practice, program transformation is not restricted to preserving the original meaning of programs and can alter the semantics in such application areas as program maintenance and evolution [80][59][13][66][16] [24][22][64][20][62] [48][27][19][7][78] [77]. Other application areas go further and translate programs from one language to another as in the development of programs by transforming formal specifications into executable code [70][68][73][74][32]. The primary motivation in all areas of program transformation is to automate programming tasks to free a programmer from manually modifying the code and instead reason about it at a higher level of abstraction.

There is a variety of tools that automate program transformation. The majority of the tools are specialized and can be classified according to the language of manipulated programs, the goal of a transformation or other factors of the environment within which a tool is intended to execute: e.g. compilers, refactoring tools and IDE plug-ins. While such tools may be efficient at a particular task, their implementations are notorious for being hard to comprehend, adapt and reuse. In contrast, program transformation systems provide dedicated constructs for traversal and rewriting of arbitrary structured entities. This enables transformation of programs of any structured language and allows one to reuse common transformations from one language to another or from one transformational goal to another with minimal changes. As a result, program transformation systems can provide (multi-)language infrastructures or domains that support a wide range of programming tasks from program refinement to compilation and refactoring.

One of the natural abstractions for expressing transformation is *rewrite rules*. They specify rewriting with input and output term patterns. Application of a rule to a term proceeds by matching the input term to the input pattern and building an output term by replacing all variables within the output pattern by the bindings produced from the input term matching. Classic term rewriting systems use rewrite rules to specify declarative rewrite relations to exhaustively apply rewrite rules everywhere in a given term [6][34][35][15]. Such systems are well-equipped to formally reason about programs in such areas as theorem-proving applications. Program transformation systems also use rewrite rules to specify the goal of rewriting, but they also provide the means for algorithmic specification of rewrite steps including what to do if a (conditional) rewrite rule fails to apply to a term[75]. This allows a programmer to specify rewrite relations that are not necessarily confluent or terminating and to build strategies for controlling the order of rule applications.

At the core of control over rewriting is conditional and sequential composition operators that define the subsequent behavior when a rule *fails to apply* to a term. Sequential composition succeeds when all rules of its ordered sequence successfully apply with individual rules applied on the outcome of a previous rule in the sequence. Conditional composition succeeds when one of the rules of the composition successfully applies to the input term with individual rules tried on the term in sequence until the first successful application. Using these constructs a programmer can define an arbitrary order of rule applications and handle failures in a *strategic* manner. For example, rewrite rules can be composed such that if the first three rules do not apply to the input term, then the next two rules should be attempted; and if all of them fail, then the input term should be left unchanged with an identity rewrite (or, instead of an identity rewrite, an exception could be raised at the point of failure).

A strategic program – a strategic composition of rewrite rules – like any other program can 'go wrong'. Programs go wrong if they do not solve the problems programmers want them to solve. In the context of controlled or strategic rewriting, a program may go wrong if it fails to modify its input term or if it modifies it incorrectly. For example, rewrite rules may be sequentially composed with a rule that always fails causing the entire program to always fail on any input term. As another example, a rule, due to its position in a strategic composition (or *strategy* for short), may never be reached during execution causing the rule to never be attempted on an input term: e.g. a rule that acts on specific terms may occur after a rule that acts on more general terms in a conditional sequence. Such scenarios cause a strategic program to not behave as intended by the programmer leading to time-intensive testing and debugging.

An effective technique for identifying if a program is behaving as expected is to understand its flow of execution [52][82]. The flow of execution in strategic rewriting can quickly become hard to analyze manually. This is due to the binary outcome of a rule application – a new term or a failure. If n number of rules are conditionally composed and applied to a term, then there are $n + 1$ possible execution paths: n possible new terms and 1 failure if all rules fail. On the other hand, if n number of rules are sequentially composed, then there are n possible failures and 1 success if all rules succeed.

The combination of conditional and sequential composition leads to multiplicative, instead of additive, growth of possible execution paths. If one conditional composition with n possible paths is sequentially composed with another conditional composition with m possible paths, then there are $n * m$ possible execution paths in the resulting strategy. In the worst case, this leads to exponential growth of execution paths as the number of such compositions grows.

In addition, the number of rule applications grows especially fast in deep rewrites – traversals of a term's structure and rewriting of all sub-terms. Since most practical transformations contain hundreds, if not thousands, of rule applications at run-time, manual control flow analysis of strategic programs becomes infeasible and automated analysis becomes a necessity.

One of most popular automated analysis methods for detection of programs that 'go wrong' is type systems. Here, a program's elements are abstracted and classified according to the kinds of values they compute. Then, a type system checks if an operation applied to some argument is defined for the argument's type. If it is, then a conservative approximation is made that the operation will succeed at run-time. While the operation may still fail at certain argument values within a type (e.g. division by zero), type checks are known to eliminate a large number of errors.

Type systems can aid the analysis of strategic rewriting by classifying the rewrite rules based on their input and output pattern types. Then, the type of a composition of rules can be computed based on the type of its constituent rules. Armed with this information, the type system can perform automated checks of whether a program can modify its inputs and if so, whether the modification produces outputs that are valid from the perspective of type abstractions.

Type analysis of strategic rewriting is substantially different from classic type analysis[60]. First, classic type analysis raises an error as soon as an operation is applied on an undefined type. Type analysis of strategies should only raise an error if none of a strategy's constituent rules, of which there could be many, is defined on an input term's type. Second, classic type analysis ignores unreachable program elements as long as such elements conform to the constraints of a language: e.g. one of the branches of a conditional may be unreachable, which is ignored as long as both branches produce terms of the same type. Type analysis of strategies is expected to identify unreachable elements of a strategy and flag them as errors.

A final notable distinction among others is that classic type analysis deals with values and types of relatively flat structure: e.g. basic types `bool` and `int`, function types and record types, whose elements are in turn either flat or have few layers of structure. In contrast, type analysis of strategies needs to deal with terms, whose structure is defined by the grammar of the term language. For most practical term languages, derivations from the root symbol of a term to the term's string involves multiple non-linear (or branching) expansions, which implies that terms are typically highly structured both in terms of breadth and depth. This adds an additional dimension of complexity, because an expressive type analysis needs to account for the terms' structures and deal with type abstractions that are as complex as the values themselves.

1.1 Contributions

The primary contributions of this research are as follows:

High-precision types Applicability of a rewrite rule to an input term depends on structural compatibility of the input term and the rule's input pattern. Previous approaches to the type analysis of term rewriting strategies have utilized sorts to assign types to rewrite rules [40]. This substantially scales down the capabilities of type-based analysis by allowing rule applications, which are statically known to always fail during execution. For example, application of a rule that acts on one constructor of a sort to a term derived from a different constructor of the same sort is well-typed, but will always fail. We enrich the analysis by including the term's structure into the term's type. Since terms may use different constructors to derive distinct strings, a term's type is now, in essence, dependent on the term's derivations. This enrichment improves expressivity by detecting at the level of types rule applications that always fail: e.g. application of a rule with an input pattern of *expr* that derives x to a term rooted in *expr* that derives y will always fail during execution and is now statically flagged as ill-typed. In other words, static analysis closely approximates, and under- or conservatively approximates when necessary, the dynamic behavior of rewriting. Consequently, automated static detection of failing rule applications allows programmers to build better rewrites or rewrites that can succeed.

Strategy types Rewrite strategies may conditionally compose rules that act on heterogeneous terms (different constructors and/or different sorts) such that if one rule fails to apply to an input term, then the next rule in the composition's sequence is attempted on the term. Thus, depending on the type of an input term, application of a conditional strategy may produce heterogeneous outputs. Application of a single rewrite rule to a term may produce a new term upon success or failure otherwise, which is another source of heterogeneity. To handle heterogeneity in rewrite strategies, we introduce union types, which are un-tagged variant types [60]. Strategic application that produces an empty union type statically indicates application failure of all constituent rules in the strategy during execution. Thus, the enriched type analysis helps programmers build strategies that can succeed.

Detailed traversal analysis Generic term traversal is a hallmark of term rewriting strategies. Having built rich term types that retain term structure and strategy types that retain types of constituent rewrite rules, it becomes possible to lift the notion of type error to term traversal strategies, which apply a strategy to sub-terms of a term. In particular, it is a type error if traversal of a term with a strategy fails for all sub-terms. Hence, the enriched type analysis helps programmers build traversal strategies that can succeed.

Among previous contributions, Lammel's work on typed generic traversals [40] is the closest related work directly addressing types within rewrite strategies. There, system S – the core of program transformation system Stratego [69] – is extended with new syntax and semantics to support types. Our contributions advance the previous work by improving the expressivity of type analysis. In other words, we can assign types and distinguish among a greater number of expressions. This in turn allows us to detect and report a greater number of errors. The importance of early detection of errors has been highlighted in [42].

The expressivity of the analysis is improved along three main directions.

1. First, our analysis uses high-precision types instead of simple types as outlined in the first contribution above.

2. Second, rewrite rules in conditional composition were previously restricted to be of the same type. This allowed one to assign a single type to the composition because the types of its operands were the same. We lift this restriction and allow the types of the composition's operands to be drawn from distinct types, which occurs commonly in practice. In particular, the type a conditional composition is now a union type as outlined in the second contribution above. Thus, our analysis allows us to assign types and check conditional compositions with heterogeneously typed rewrite rules, which were previously viewed as type errors.

3. Third, term traversals were previously assigned one of two type schemes: TP

(or type-preserving) for traversals that modify a term but preserve its type and $TU(T)$ (or type-unifying for some type T) for traversals that extract information from a term or summarize it into a single value of type T. We assign types to term traversals that depend on the type of its argument strategy. This enables detection of application of a term traversal to a term that does not have a sub-term applicable for the argument strategy. Thus, our analysis can detect a greater number of errors.

A final notable distinction is that we analyze programs of strategic rewriting language TL that is fundamentally different from that of system S. In particular, handling of application failure in TL is refined by leaving the input term unchanged instead of raising a run-time exception. This results a substantially different semantic behavior that requires a richer static view of application results.

Structure The presentation of type analysis is structured as follows. In the remainder of this chapter we present several motivating examples and summarize the kinds of errors that commonly occur in rewrite strategies. In Chapter 2, we summarize type systems as a syntax-directed method of proving absence of undesirable run-time errors and present two examples of type analysis of small object languages. Chapter 3 presents an overview of transformation language TL that is a representative transformation language supporting all of the primary abstractions of rewrite strategies. Chapter 3 concludes with the discussion of related transformation systems and languages including their analysis capabilities. Chapters 4 and 5 present the main contributions of this research. In Chapter 4, we formally define the core aspects of type analysis including the definition of types, typing contexts, typing relation and analysis of the standard rewrite strategy features such as patterns, matching, rewrite rules and combinators. Chapter 5 presents extensions of the core system by incorporating analysis of non-standard features of TL such as invocation of functional programming from within a conditional rewrite rule as well as higher-order rules and compositions. Chapter 5 concludes with the current summary of the analysis of term traversals, whose full analysis remains as future work. Throughout all of the presentation of type analysis, we provide fully type-checked examples to demonstrate the concrete results of type analysis. Chapter 6 presents the aspects of an ML-based implementation of the declarative type analysis presented in the previous two chapters. The goal of this chapter is to highlight key components of the implementation. Finally, Chapter 7 concludes and presents the limitations and aspects of type analysis that remain as future work.

1.2 Motivating examples

To observe common pitfalls and errors in the programming of rewrite strategies in concrete terms, let us discuss several examples of program transformation. A type system capable of static detection of such errors can substantially improve validation of rewrite strategies.

Strategy	Name	Purpose
$\mathtt{t}_x$	meta- or schema variable	ranges over all derivations from $\mathtt{t}$
$\mathtt{t}[\![\mathtt{t}_1\mathtt{t}_2\ldots\mathtt{t}_n]\!]$	pattern expression	specifies terms rooted in $\mathtt{t}$
$\mathtt{t}_i \rightarrow \mathtt{t}_o$	rewrite rule	input-output pattern pair
`id`	identity strategy	applies to and leaves any term unchanged
$\mathtt{s}_1 \lessdot\!\!+ \mathtt{s}_2$	conditional composition	apply $\mathtt{s}_2$ if $\mathtt{s}_1$ fails
$\mathtt{s}_1 \lessdot\!* \mathtt{s}_2$	(strict) sequential composition	apply $\mathtt{s}_2$ if $\mathtt{s}_1$ succeeds
$\mathtt{s}_1 \lessdot\!; \mathtt{s}_2$	non-strict seq. composition	apply $\mathtt{s}_2$ to any output of $\mathtt{s}_1$
`map(s)`	immediate sub-term traversal	apply `s` to all immediate sub-terms

Figure 1.1: Representative strategic rewriting constructs

Preliminaries Let us adopt the constructs of Figure 1.1 as a representative notation for expressing strategic rewriting. A meta-variable ranging over all derivations from $\mathtt{t}$ is denoted by $\mathtt{t}_x$ with an alphanumeric subscript x. To denote a pattern in the concrete syntax of the term language, we use $\mathtt{t}[\![\mathtt{t}_1\ \mathtt{t}_2\ \ldots\ \mathtt{t}_n]\!]$, which describes a tree, where $\texttt{t}$ is the root symbol of the term and $\texttt{t}_i$'s are tree leaves that represent meta-variables or concrete tokens. Rewrite rules map input patterns to output patterns: $\texttt{t}_i \quad \rightarrow \texttt{t}_o$. The strategy `id` is a polymorphic strategy that applies to any term and leaves it unchanged. Combinators determine the outcome of a composition if the first strategy of the composition fails: conditional combinator ($<\!\!+$) returns the outcome of the second strategy, sequential combinator ($<\!\!*$) returns the failure, and non-strict sequential combinator ($<\!;$) applies the second strategy on the outcome of the first strategy whether it succeeded or not. Finally, to enable traversal of sub-terms, combinator `map` applies its argument to all immediate sub-terms of an input term.

Failure of a rule and in general a strategy to apply to a term can be viewed from at least two different perspectives. In the first perspective, which we call *exception-based*, failure creates a run-time exception, which terminates the execution, or it creates a distinguished meta-term (e.g. $\uparrow$). In the second *identity-based* perspective, failure is treated in a more refined manner by treating the results of an application as a tuple of an output term and a boolean constant indicating the result of an application[79]. Thus, if f is a strategy that always fails and t is an input term, then the result of application $f\ t$, in the exception-based framework is `error` or $\uparrow$, and in the identity-based framework is $\langle t, false\rangle$.

While failure to apply is more directly observable in exception-based frameworks, identity-based behavior is at times built-in as a primitive abstraction: e.g. `try(s)`, which is the same as $\texttt{s}\ <\!\!+\ \texttt{id}$. However, such approximations do not produce the same behavior, because $\texttt{try(f)}\ <\!\!+\ \texttt{s}$ in an exception-based framework will always succeed and leave the input term unchanged, while $\texttt{f}\ <\!\!+\ \texttt{s}$ in an identity-based framework will return the results of `s`. We adopt the refined approach and view application failure from the identity-based perspective.

The small set of strategic constructs summarized in Figure 1.1 combined with recursion can be used not only to control rewriting but also to define deep traversal and rewriting of terms. For example, to rewrite every node in a term's tree we could define the following top-down traversal

```
topdown(s) = s  <; map(topdown(s))
```

where strategy `s` is applied to the root node before the recursive descent into its sub-terms.

Repetitive application of a strategy to a term to compute its normal form can be expressed by the following combination of the primitive constructs:

```
repeat(s) = s  <* (repeat(s)  <+ id)
```

where strategy `s` is recursively applied to a term until the first failed application, which stops the recursion by invoking `id` on the normalized term.

1.2.1 Boolean expressions

Consider the language of simple boolean expressions consisting of the boolean constants and an 'if-then-else' conditional. Figure 1.2 summarizes the grammar of the language and its evaluation steps[60].

Syntax				*Evaluation*	
t	::=		*terms:*	$if\ true\ then\ t_2\ else\ t_3 \to t_2$	(E-IfTrue)
		v	*value*		
		if t then t else t	*conditional*	$if\ false\ then\ t_2\ else\ t_3 \to t_3$	(E-IfFalse)
v	::=		*values:*		
		true	*true*	$\dfrac{t_1 \to t_1'}{if\ t_1\ then\ t_2\ else\ t_3 \to if\ t_1'\ then\ t_2\ else\ t_3}$	(E-If)
		false	*false*		

Figure 1.2: Booleans

Suppose the expressions of this language need to be simplified using rewrite strategies.[1] The evaluation steps given in Figure 1.2 in the form of declarative inference rules can be directly encoded as rewrite rules:

$$\begin{aligned}
\texttt{ifTrue} &= \texttt{t}[\![\texttt{if true then t}_2\texttt{ else t}_3]\!] \;\rightarrow\; \texttt{t}_2 \\
\texttt{ifFalse} &= \texttt{t}[\![\texttt{if false then t}_2\texttt{ else t}_3]\!] \;\rightarrow\; \texttt{t}_3 \\
\texttt{ifCongruence} &= \texttt{t}[\![\texttt{if t}_1\texttt{ then t}_2\texttt{ else t}_3]\!] \;\rightarrow \\
& \ \texttt{t}[\![\texttt{if step(t}_1\texttt{) then t}_2\texttt{ else t}_3]\!]
\end{aligned}$$

Note that the congruent evaluation rule (E-If) declares an intermediate computation of t'_1, which is obtained by (possibly recursive) application of the three rules. This is captured by $\texttt{step(t}_1\texttt{)}$ in rule `ifCongruence`, which produces t'_1 in the rule's output pattern. Since rewrite strategies need to algorithmically specify the order of application of rewrite rules, our goal is to program the strategy `step`, which defines the composition and recursion of the three basic rewrite rules.

Suppose a first attempt toward a solution is a sequential composition of the rules: e.g.

$$\texttt{step' = ifTrue} \ \lessdot\!\ast\ \texttt{ifFalse}$$

However, during testing this composition would actually fail to apply to simple terms like `if true then true else false`, which can clearly be rewritten to `true` in one step. This would occur because sequential composition implies that the input term of the second strategy must be compatible with the output term of the first strategy. Therefore, meta-variable $\texttt{t}_2$ of `ifTrue` would become bound to the input term of

[1] Appendix A provides the implementation of the rewriting of booleans in *TL*.

`ifFalse` and the actual pattern of `ifTrue` would become

$$\begin{array}{l} \texttt{t}[\![\ \texttt{if true then} \\ \qquad\qquad\qquad\qquad \texttt{if false then t}_{2'}\ \texttt{else t}_{3'} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else t}_{3}]\!] \end{array}$$

In effect, we determined the most general term structure to which the strategy `ifTrue` $\mathrel{<\!\!\ast}$ `ifFalse` could be applied.

Note that it is not necessary to execute the composition on actual inputs to observe how the composition affects the patterns of constituent rewrite rules. This is where a type system can intervene and compute the types of the patterns, rules and strategies. For every application in a program, the type system can check if the input type of an operation is compatible with the type of the operation's argument. Such incompatibilities can be detected statically and fixed by a programmer prior to execution. At a minimum, the type system can act as a *programmer's assistant* in computing types of strategies. A programmer can then compare the computed type of a strategy against the intended type and make any necessary code changes, if the two types do not match.

In the example above, the calculated *type* of the strategy is

$$\texttt{t}[\![\texttt{if true then if false then t}_{2'}\ \texttt{else t}_{3'}\ \texttt{else t}_{3}]\!] \ \rightarrow \texttt{t}_{3'}$$

Since `ifTrue` $\mathrel{<\!\!\ast}$ `ifFalse` is only applicable to terms that are instances of or more specific than either one of the constituent rule's inputs, it clearly does not capture the intent of simplifying both kinds of possible terms. A more appropriate composition of these rules is the conditional composition. Together with the congruent rewrite rule, the correct strategy is

```
step = ifTrue  <+ ifFalse  <+ ifCongruence
```

This composition encodes a one-step reduction. To extend it to a multi-step exhaustive reduction, the strategy is wrapped inside a normalizing loop:

```
main = repeat(step)
```

1.2.2 Arithmetic expressions

Suppose the language of boolean expressions is extended with natural numbers and their operations. Figure 1.3 summarizes the new parts of this extension[60].

New syntactic forms

t	::=	...	*terms:*
		succ t	*successor*
		pred t	*predecessor*
		iszero t	*zero test*
v	::=	...	*values:*
		nv	*numerical value*
nv	::=		*numerical values:*
		0	*zero value*

New evaluation rules

$$\frac{t_1 \rightarrow t_1'}{succ\ t_1 \rightarrow succ\ t_1'} \qquad \text{(E-Succ)}$$

$$pred\ 0 \rightarrow 0 \qquad \text{(E-PredZero)}$$

$$pred\ succ\ t_1 \rightarrow t_1 \qquad \text{(E-PredSucc)}$$

$$\frac{t_1 \rightarrow t_1'}{pred\ t_1 \rightarrow pred\ t_1'} \qquad \text{(E-Pred)}$$

$$iszero\ 0 \rightarrow true \qquad \text{(E-IsZeroZero)}$$

$$iszero\ succ\ t_1 \rightarrow false \qquad \text{(E-IsZeroSucc)}$$

$$\frac{t_1 \rightarrow t_1'}{iszero\ t_1 \rightarrow iszero\ t_1'} \qquad \text{(E-IsZero)}$$

Figure 1.3: Arithmetic expressions

Similar to rewrite strategies for boolean expressions, the evaluation rules of arithmetic expressions could be directly encoded into rewrite rules as follows:[2]

$$
\begin{aligned}
\texttt{predZero'} &= \texttt{t}[\![\texttt{pred 0}]\!] \rightarrow \texttt{v}[\![\texttt{0}]\!] \\
\texttt{predSucc} &= \texttt{t}[\![\texttt{pred (succ t}_1\texttt{)}]\!] \rightarrow \texttt{t}_1 \\
\texttt{isZeroZero'} &= \texttt{t}[\![\texttt{iszero 0}]\!] \rightarrow \texttt{v}[\![\texttt{true}]\!] \\
\texttt{isZeroSucc'} &= \texttt{t}[\![\texttt{iszero (succ t}_1\texttt{)}]\!] \rightarrow \texttt{v}[\![\texttt{false}]\!] \\
\texttt{succCongr} &= \texttt{t}[\![\texttt{succ t}_1]\!] \rightarrow \texttt{t}[\![\texttt{succ step(t}_1\texttt{)}]\!] \\
\texttt{predCongr} &= \texttt{t}[\![\texttt{pred t}_1]\!] \rightarrow \texttt{t}[\![\texttt{pred step(t}_1\texttt{)}]\!] \\
\texttt{isZeroCongr} &= \texttt{t}[\![\texttt{iszero t}_1]\!] \rightarrow \texttt{t}[\![\texttt{iszero step(t}_1\texttt{)}]\!] \\
\texttt{step} &= \texttt{ifTrue} \mathbin{\lhd\!\!\!+} \texttt{ifFalse} \mathbin{\lhd\!\!\!+} \texttt{ifCongruence} \mathbin{\lhd\!\!\!+} \\
& \quad \texttt{predZero} \mathbin{\lhd\!\!\!+} \texttt{predSucc} \mathbin{\lhd\!\!\!+} \texttt{isZeroZero} \mathbin{\lhd\!\!\!+} \texttt{isZeroSucc} \mathbin{\lhd\!\!\!+} \\
& \quad \texttt{succCongr} \mathbin{\lhd\!\!\!+} \texttt{predCongr} \mathbin{\lhd\!\!\!+} \texttt{isZeroCongr}
\end{aligned}
$$

However, during testing the program would fail to completely simplify some terms: e.g. `iszero pred 0`. This would occur because some of the rewrite rules change the type of the root term: e.g. `predZero'` rewrites terms derived from `t` to terms derived from `v`. Because of this, congruent rules would fail to simplify the sub-terms, which would leave the input term unchanged or incompletely simplified.

A type system could detect that some rules in the composition are type-changing and *raise a warning* to the programmer.

[2] Appendix B provides the implementation of the rewriting of arithmetic expressions in *TL*.

The failure-causing rewrite rules can be corrected as follows:

$$\begin{aligned}
\texttt{predZero} &= \texttt{t}[\![\texttt{pred 0}]\!] \;\to\; \texttt{t}[\![\texttt{0}]\!] \\
\texttt{isZeroZero} &= \texttt{t}[\![\texttt{iszero 0}]\!] \;\to\; \texttt{t}[\![\texttt{true}]\!] \\
\texttt{isZeroSucc} &= \texttt{t}[\![\texttt{iszero (succ t}_1\texttt{)}]\!] \;\to\; \texttt{t}[\![\texttt{false}]\!]
\end{aligned}$$

1.2.3 Lambda calculus

Consider the language of lambda-calculus, which is at the core of computation in functional programming languages. Suppose the evaluation of expressions in this language needs to be accomplished using rewrite strategies.[3] Figure 1.4 lists the grammar and declarative evaluation rules of the language[60].

Syntax

t	::=		*terms:*
		x	*variable*
		v	*value*
		t t	*application*
v	::=		*values:*
		λx. t	*abstraction value*

Evaluation

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2'} \qquad \text{(E-App2)}$$

$$\lambda x.\ t_1\ v_2 \to [x \mapsto v_2]\ t_1 \qquad \text{(E-AppAbs)}$$

Figure 1.4: Lambda calculus

Note that evaluation rules are mutually exclusive by ensuring that function application or beta-reduction takes place only if both terms of an application are reduced to values (rules (E-App1) and (E-App2)). If so, all occurrences of a lambda-bound variable x are replaced by the argument v_2 everywhere in the body t_1 (rule (E-AppAbs)). In addition, we are simplifying and ignoring possible variable name conflicts: i.e. alpha-conversion to avoid variable capture.

[3] Appendix C provides the implementation of the rewriting of lambda expressions in *TL*.

A direct translation of evaluation rules to rewrite rules produces the following rewrite strategies:

$$
\begin{array}{rcl}
\texttt{app1'} & = & \texttt{t}[\![\texttt{t}_1\ \texttt{t}_2]\!] \ \rightarrow \texttt{t}[\![\texttt{step'(t}_1\texttt{)}\ \texttt{t}_2]\!] \\
\texttt{app2'} & = & \texttt{t}[\![\texttt{v}_1\ \texttt{t}_2]\!] \ \rightarrow \texttt{t}[\![\texttt{v}_1\ \texttt{step'(t}_2\texttt{)}]\!] \\
\texttt{appAbs} & = & \texttt{t}[\![\lambda\ \texttt{x}_1\texttt{.}\ \texttt{t}_1\ \texttt{v}_1]\!] \ \rightarrow \texttt{topdown(x}_1 \rightarrow \texttt{v}_1\texttt{)}\ \texttt{t}_1 \\
\texttt{step'} & = & \texttt{app1'} \ \mathbin{<\!\!\!+}\ \texttt{app2'} \ \mathbin{<\!\!\!+}\ \texttt{appAbs}
\end{array}
$$

However, during testing this would actually fail to modify any input term. This is due to an incorrect ordering of rewrite rules in conditional composition. The rule `app1'` has an input pattern that is actually more general than the other two rules's patterns, because symbol `t` derives symbol `v` and thus pattern $\texttt{t}[\![\texttt{t}_1\ \texttt{t}_2]\!]$ is more general than the other patterns. This will cause the first rule of the composition `step'` to always apply and never modify an input term.

For example, suppose the input term is $\lambda\texttt{a.a}\ \lambda\texttt{b.b}$, which is supposed to be evaluated to just $\lambda\texttt{b.b}$. During program execution, the first rule of the conditional composition – `app1'` – will be attempted on the input, which will lead to the invocation $\texttt{step'(}\lambda\texttt{a.a)}$. This will fail because none of the other rule's input patterns match the argument, which will leave the argument unchanged at $\lambda\texttt{a.a}$. The result will be placed in the output pattern and the rule `app'` will succeed and produce $\lambda\texttt{a.a}\ \lambda\texttt{b.b}$, which is the same as the original input term. Hence, the program will fail to modify this input term and in fact any other input term.

As another example, consider the strategy `id <+ s`. This strategy always succeeds with an identity rewrite because the first strategy is applicable to any input term causing the other strategy `s` to never have a chance of applying to a term. Such composition is clearly also an error.

In general, a conditional sequence of strategies should be ordered such that more generic strategies (i.e. strategies with input patterns that are more generic) should be ordered after less generic strategies to allow every strategy in the sequence a chance of rewriting an input term.

A type system can aid the programmer in the correct ordering of strategies by calculating the types of constituent rewrite rules. If a situation is detected where a more generic strategy occurs before a less generic strategy in a conditional sequence, in other words it is overshadowed, then a type error can be raised to prompt a programmer to re-order the sequence or perform other corrections.

Overshadowing

A strategy, whose constituent rule is unreachable because it is overshadowed by rules earlier in the strategy's sequence, is an error.

In the example, a possible correction is to re-order the strategies in the following sequence:

$$\texttt{step} \;=\; \texttt{appAbs} \;\lhd\!\!+\; \texttt{app2'} \;\lhd\!\!+\; \texttt{app1'}$$

Another variant of a correction is to make all input patterns mutually exclusive: e.g.

$$\begin{aligned}\texttt{app1} &= \texttt{t}\llbracket(\texttt{t}_{11}\ \texttt{t}_{12})\ \texttt{t}_{2}\rrbracket \;\rightarrow\; \texttt{t}\llbracket\texttt{step}(\texttt{t}_{11}\ \texttt{t}_{12})\ \texttt{t}_{2}\rrbracket\\ \texttt{app2} &= \texttt{t}\llbracket\texttt{v}_{1}\ (\texttt{t}_{21}\ \texttt{t}_{22})\rrbracket \;\rightarrow\; \texttt{t}\llbracket\texttt{v}_{1}\ \texttt{step}(\texttt{t}_{21}\ \texttt{t}_{22})\rrbracket\end{aligned}$$

Then, when both terms are completely simplified to values and the pattern of application becomes $\texttt{t}\llbracket\texttt{v}_1\ \texttt{v}_2\rrbracket$, the base case expressed by rule `appAbs` will be applied on the term to perform the actual function application or beta-reduction.

1.3 Summary of errors and other issues

The examples above illustrate both the strategic rewriting approach to program transformation as well as the errors that can occur even in small transformational tasks. A type system geared toward analysis of strategic programs can not only act as a programmer's assistant in the computation of strategy types and raise warnings when undesirable conditions are detected, but also catch errors that can occur in strategic compositions of rewrite steps. An example of one kind of error is overshadowing, where a strategy is unreachable due to its position in a conditional sequence. There are several other commonly occurring kinds of errors in programming of strategies and we summarize them next.

Incorrect composition Composition of strategies needs to satisfy the semantics of combinators used in the composition. In strict sequential composition ($<\!\!*$), output of the first strategy needs to be compatible with the input of the second strategy. For example, the following composition

$$\mathtt{badSequence} \;=\; \mathtt{t}[\![\mathtt{pred\ 0}]\!] \;\rightarrow\; \mathtt{t}[\![\mathtt{0}]\!] \;\;<\!\!*\;\; \mathtt{t}[\![\mathtt{pred\ succ\ t_1}]\!] \;\rightarrow\; \mathtt{t_1}$$

has no chance of succeeding because the second strategy will never apply to the output of the first. This is clearly an error.

Infeasible sequence	A strict sequential composition, where one of the strategies always fails.

Related to the category of errors due to incorrect composition is a conditional composition strategy, which given an input term, is not defined for the term's type. In other words, all constituent strategies fail to apply to the term, which may occur if a programmer did not account for some of the possible input term kinds. If none of the enclosing strategies can handle the input term, then a global application failure occurs and the input term is left unchanged. For example, the following application

$$\texttt{badChoiceApp} \;=\; (\texttt{id}[\![\mathrm{x}]\!] \;\rightarrow\; \texttt{id}[\![\mathrm{y}]\!] \;\mathrel{\lhd\!\!\!+}\; \texttt{id}[\![\mathrm{y}]\!] \;\rightarrow\; \texttt{id}[\![\mathrm{z}]\!])\;\texttt{id}[\![\mathrm{z}]\!]$$

will always fail at run-time because none of the constituent rewrites in the conditional composition are prepared to apply to the input term.

Infeasible conditional strategy application

Application of a conditional composition that always fails.

A type system can statically determine such errors allowing a programmer to fix them prior to execution and testing.

Match equations Rewrite strategies often use pattern match equations to either bind variables to values or to compare values bound to variables: $\texttt{t}[\![\texttt{t}_1\ \texttt{t}_2\texttt{...t}_n]\!]\ \texttt{==}\ \texttt{s}$. For a match equation to succeed, the terms on both sides of the equation need to be compatible. While some errors are easy to detect

$$\texttt{t}_1\ \texttt{==}\ \texttt{v}[\![\texttt{true}]\!]$$

where the roots of the two terms are not the same, other errors are not so obvious

$$\texttt{t}_1\ \texttt{==}\ \texttt{<long and complex strategy>}$$

because there may be multiple potential match candidates.

Infeasible pattern match

Pattern match of terms, which cannot succeed.

A type system can aid the programmer in determining compatibility of patterns in match equations.

Term traversals In deep rewrites of a term using term traversals, it is possible for a strategy to always fail because the input term does not have any sub-terms compatible with the strategy. For example, if symbol `expr` does not derive symbol `stmt` then traversal of a term rooted in `expr` with a strategy, whose input pattern is rooted in `stmt` has no chance of succeeding and is clearly an error:

$$\texttt{badTraversal} \;=\; \texttt{topdown(stmt}_1 \;\rightarrow\; \texttt{stmt}_1\texttt{)}\ \texttt{expr}_1$$

Unviable traversals

Traversal of a term with a strategy, which cannot succeed for any sub-term of the term.

A type system can inspect the structure of an input term and determine if the input pattern of a traversal strategy is among the possible sub-terms of the input term.

Free variables Rewrite rules and strategies typically cannot reference unbound or free variables. A variable occurs free if it is not bound by either an input pattern or other match equations within a rule or a strategy. For example, variable $\texttt{t}_1$ occurs free below

$$\texttt{freeVariable} \;=\; \texttt{t}[\![\texttt{if true then t}_2\ \texttt{elset}_3]\!] \;\rightarrow\; \texttt{t}_1$$

In practice, free variables are often the result of typographical errors: e.g. they occur often in program maintenance, when existing code is modified, and with less frequency in regular program development. A type system can automatically analyze program variables and flag those occurring free.

Program maintenance A type system can be an invaluable tool in program maintenance. For example, if a programmer wants to change or refactor the grammar of the term language, he will not need to search by hand all the places in a large strategic program where the code involving this grammar change needs to be fixed. All of these sites become type-inconsistent and can be enumerated simply by running the type-checker and examining the points of type analysis failure.

Equational Reasoning Equational reasoning can be used to refactor strategic programs, where strategies of one form are replaced with behaviorally equivalent strategies of another form. Refactoring could be used to increase readability or efficiency of strategic programs. For example, consider the following strategies

$$\begin{aligned}
r_1 &= \mathit{stmt}[\![x = \mathit{expr}_1;]\!] \rightarrow \mathit{stmt}[\![\ldots]\!] \\
r_2 &= \mathit{stmt}[\![y = \mathit{expr}_1;]\!] \rightarrow \mathit{stmt}[\![\ldots]\!] \\
s_1 &= r_1 <\!; r_2 \\
s_2 &= r_1 <\!\!\!+ r_2
\end{aligned}$$

Strategy s_2 is equivalent to s_1 because given a term only one of the rules r_1 and r_2 has a chance of succeeding since an input term cannot match both x and y within assignment statements at the same time. Since conditional composition carries fewer constraints (less control-flow information) than sequential composition and because it is more readable and efficient, conditional composition is the preferred choice when the two kinds of composition are equivalent. A type system can analyze strategies and suggest a more optimized re-factoring of the strategies using equational reasoning.

Language-specific errors A transformation language used to program rewrite strategies can have certain conditions, which can be statically checked by a type system. For example, a language might not allow duplicate strategy declarations

$$
\begin{array}{rcl}
\texttt{s} & = & \ldots \\
 & \ldots & \\
\texttt{s} & = & \ldots
\end{array}
$$

Since type systems perform name analysis as part of their standard analysis routine, duplicate strategies can easily be checked and flagged.

As another example, a language may require existence of a distinguished strategy declaration, which drives the entire execution: e.g. `main`. A type system can flag programs that do not define a `main` strategy. These and other language-specific conditions can be performed by a type system during the standard type analysis routine.

Chapter 2

Type Systems

Type system is one of the most popular and well-established formal methods for detection of errors:

> *A type system is a tractable syntactic method for proving absence of certain program behaviors by classifying phrases according to the kinds of values they compute [60].*

Type systems use a collection of syntax-directed rules for computing types of program phrases or terms based on the types of constituent elements. If a rule can be applied on term t to calculate its type T, then the term is said to be well-typed, which is denoted as $t : T$. If no rule can be applied to a term, then the term is ill-typed and flagged as an error.

The classic notation for describing the typing rules is the inference rule notation (also known as Gentzen's notation), where a rule's premises and its conclusion are separated by a horizontal bar. If there are no premises, then the rule has an empty or no bar. For example, the inference rule

$$\frac{t_1 : Bool \qquad t_2 : T \qquad t_3 : T}{if\ t_1\ then\ t_2\ else\ t_3\ :\ T}$$

states that for a conditional term to be well-typed, the guard of a conditional must have a boolean type and that both branches of the conditional must have the same type T. Other terms that do not fit these conditions are ill-typed: e.g.

- `if 0 then 0 else 1`, because the conditional's guard is not a boolean term,
- `if true then 1 else false`, because the conditional's branches' types are not the same.

One of the desirable properties of a type system is its soundness or that well-typed terms should not "go wrong" during execution. In other words, if a type system classifies a term to be well-typed, then the evaluation of the term at run-time should not produce any type-related errors. The rules of what it means to go wrong or in other words the definition of a run-time error vary from one language to another. The classic notion of a run-time error is a "stuck state", where no further evaluation rule applies to the term (i.e. it is in normal form), yet the term is not a valid value. For example, in a language that does not allow addition with booleans, term `5 + true` is in normal form, yet it is neither a boolean value nor an integer value. A sound type system will not assign a type to this term.

Soundness of a type system is typically established by progress and preservation theorems [60]:

Progress A well-typed term is not stuck: either it is a valid value or it is reducible according to the evaluation rules.

Preservation A well-typed and reducible term, when evaluated, produces another well-typed term.

If both of these theorems hold, then a terminating evaluation of a well-typed term is guaranteed to produce a valid value. As a result, a sound type system can eliminate all run-time errors detectable at the abstraction of types.

A property that is dual to soundness is completeness, where a type system does not reject as ill-typed any valid term. Toward this end, type systems typically err on the side of caution to remain tractable/efficient, which leads to conservativeness in the type analysis. For example, terms like

```
if <long and complex test> then 5 else <type error>
```

are typically rejected by type systems as ill-typed even if the conditional's guard will always evaluate to true, because precise type analysis of the guard may be intractable.

To provide an intuition for the soundness and completeness of a type system, suppose set C represents programs that are correct and set T represents programs that are well-typed. Then, there are four possible relations between the two sets (also known as the Venn's diagrams):

1. $C \in T$: In this case the type system is complete, but not sound
2. $T \in C$: The type system is sound, but not complete
3. $C \cap T = A, A \neq C, A \neq T$: The type system is neither sound nor complete
4. $C = T$: The type system is both sound and complete.

Cases 2 and 4 are the most desirable properties that a type system can have with case 4 being the ideal, but hard to attain due to tractability reasons.

The tradeoff between tractability/conservativeness and precision/expressivity is inherent in the design of any type system. Improvement of the precision of type-based analysis is one of the main goals of type systems research. For example, dependent types have been recently proposed to detect array accesses beyond array bounds, which is not detectable using simple types. As the computing power and type systems research innovations grow, type systems are expected to become not only sound but also more complete.

2.1 Type Analysis using Rewrite Strategies

To observe the calculation of types and type-checking by a type system, let us discuss the implementations of a type system for two languages: arithmetic expressions and lambda-calculus. In both cases, the implementations use rewrite strategies to analyze a term and rewrite the term to the term's type. The result of type analysis is the term's type, if it is well-typed, or term `Abort` otherwise. The goal of these examples is not to discuss potential sources of errors in programming rewrite strategies, but to illustrate the type analysis in operation.

2.1.1 Typed arithmetic expressions

Consider the language of arithmetic expressions summarized in Figure 1.3. Figure 2.1 defines the typing rules of this language [60].

New syntactic forms

T ::=		*types:*
	$Bool$	*booleans*
	Nat	*nat. numbers*

New typing rules

$$true : Bool \qquad \text{(T-True)}$$

$$false : Bool \qquad \text{(T-False)}$$

$$\frac{t_1 : Bool \quad t_2 : T \quad t_3 : T}{if\ t_1\ then\ t_2\ else\ t_3 : T} \qquad \text{(T-If)}$$

New typing rules (cont'd)

$$0 : Nat \qquad \text{(T-Zero)}$$

$$\frac{t_1 : Nat}{succ\ t_1 : Nat} \qquad \text{(T-Succ)}$$

$$\frac{t_1 : Nat}{pred\ t_1 : Nat} \qquad \text{(T-Pred)}$$

$$\frac{t_1 : Nat}{iszero\ t_1 : Bool} \qquad \text{(T-IsZero)}$$

Figure 2.1: Typing rules for arithmetic expressions

Rewrite strategies with their inherent pattern-matching capabilities are well-equipped for type analysis of arithmetic expressions.[1] To support the rewriting of terms to types, we extend the term language to include the terms *Bool* and *Nat* to represent the types of well-typed terms and *Abort* to represent the terms that contain

[1] Appendix D provides the implementation of type-checking of arithmetic expressions in *TL*.

type errors:

$$
\begin{array}{rcl}
t ::= & \ldots & \textit{terms:} \\
 & ty & \textit{type of a term} \\
ty ::= & & \textit{types:} \\
 & \mathit{Bool} & \textit{type of booleans} \\
 & Nat & \textit{type of naturals} \\
 & \mathit{Abort} & \textit{type of ill-typed terms}
\end{array}
$$

The encoding of typing rules into rewrite strategies is similar to the encoding of evaluation rules as discussed in Section 1.2.2. In particular, the choice of a rewrite rule to apply to the input term depends on the syntactic structure of the term and the premises of inference rules are checked by (possibly recursive) rewriting of subterms. Figure 2.2 summarizes the rewrite strategies that implement the type analysis. Some of the rewrite rules contain additional constraints in the body of an *if*-block. These constraints either bind local variables (=) or perform a comparison of two ground terms (==). A rule with an *if*-block succeeds if the boolean composition of all constraints within the block succeeds.

Note that the `main` strategy contains a post-processing step that checks if the normalized term $\mathtt{t}_1$ is a valid type, otherwise term `Abort` is produced. This enables the type-checker to signal a type error if all of the rules in the conditional composition `step` fail to apply. Therefore, type analysis of well-typed terms produces either `Bool` or Nat and analysis of ill-typed terms produces `Abort`.

2.1.2 Simply typed lambda calculus

Let us now consider the lambda calculus, whose type analysis is more involved due to the presence of term variables bound by lambda abstractions. Figure 2.3 defines

```
  tTrue    =   t⟦true⟧  → t⟦Bool⟧
 tFalse    =   t⟦false⟧  → t⟦Bool⟧
    tIf    =   t⟦if $t_1$ then $t_2$ else $t_3$⟧  → $t_{then}$
                 if {t⟦Bool⟧ == step($t_1$) and
                     $t_{then}$ = step($t_2$) and
                     $t_{else}$ = step($t_3$) and
                     $t_{then}$ == $t_{else}$
                 }

  tZero    =   t⟦0⟧  → t⟦Nat⟧
  tSucc    =   t⟦succ $t_1$⟧  → t⟦Nat⟧ if { t⟦Nat⟧ == step($t_1$)}
  tPred    =   t⟦pred $t_1$⟧  → t⟦Nat⟧ if { t⟦Nat⟧ == step($t_1$)}
tIsZero    =   t⟦iszero $t_1$⟧  → t⟦Bool⟧ if { t⟦Nat⟧ == step($t_1$)}

   step    =   tTrue  <+ tFalse  <+ tIf  <+
               tZero  <+ tSucc  <+ tPred  <+ tIsZero

   main    =   $t_{in}$  → $t_{out}$
                 if { $t_1$ = ((repeat step) $t_{in}$) and
                      ((($t_1$ == t⟦Bool⟧ or $t_1$ == t⟦Nat⟧) and $t_{out}$ = $t_1$)
                      or $t_{out}$ = t⟦Abort⟧)
                 }
```

Figure 2.2: Type analysis of arithmetic expressions using rewrite strategies

the typing rules of the language [60].

Due to the need to remember the type of an abstraction's variable in the analysis of the abstraction's body, the typing relation now changes from a binary to a ternary relation $\Gamma \vdash t : T$, where Γ is the typing context or environment that holds a set of bindings of variables and their types. Thus, to determine the type of an abstraction, rule *(T-Abs)* extends the typing context with a type binding for the abstraction's variable prior to the calculation of the type of the abstraction's body. Application of term t_1 to term t_2 is well-typed if term t_1 has a function type, whose input type is the same as the type of term t_2 as expressed by rule *(T-App)*.

Despite the addition of typing contexts to the typing relation to remember a variable's type binding, rewrite strategies can perform type analysis of lambda expres-

New syntactic forms

v	$::=$		*values:*
		$\lambda x : T.\ t$	*abstraction value*
T	$::=$		*types:*
		$T \to T$	*type of functions*
		A	*base types*
Γ	$::=$		*contexts:*
		$\varnothing$	*empty context*
		$\Gamma, x : T$	*variable binding*

Typing rules

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-Var)}$$

$$\frac{\Gamma, x : T \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.\ t_2 : T_1 \to T_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \qquad \text{(T-App)}$$

Figure 2.3: Simply typed lambda calculus

sions.[2] For this purpose, analysis of an abstraction's body is preceded by distribution of type information in the body through rewriting of all occurrences of the variable by its type. Then, the pre-processed term can be examined on its own without any outside references to typing contexts. Figure 2.4 summarizes this implementation.

As before, the term language is extended with terms that represent types:

$t ::=$	$\ldots$	*terms:*
	ty	*type of a term*
$ty ::=$		*types:*
	$ty \to ty$	*type of functions*
	A	*base types*
	$Abort$	*type of ill-typed terms*

Note that rule `tAbs` in Figure 2.4 performs a top-down traversal of an abstraction's body `t`$_2$ to rewrite all occurrences of `x`$_1$ by its type `ty`$_1$. This allows the implementation to inline the inference rule *(T-Var)* and the extension of the typing context in the inference rule *(T-Abs)*. The remaining elements of the implementation follow directly from the typing rules.

[2] Appendix E provides the implementation of type-checking of lambda calculus expressions in *TL*.

```
tAbs        =       t⟦λ $x_1$:  $ty_1$.  $t_2$⟧  → t⟦$ty_1$ → $ty_2$⟧
                      if {$t_3$ = (topdown($x_1$  → $ty_1$) $t_2$) and
                          t⟦$ty_2$⟧ = step($t_3$) and
                      }

tApp        =       t⟦$t_1$ $t_2$⟧  → t⟦$ty_2$⟧
                      if {t⟦$ty_1$ → $ty_2$⟧ = step($t_1$) and
                          $ty_1$ == step($t_2$)
                      }

step        =       tAbs  $\lhd\!+$ tApp

main        =       $t_{in}$  → $t_{out}$
                      if { $t_1$ = ((repeat step) $t_{in}$) and
                           (($t_1$ == $t_{in}$ and $t_{out}$ = t⟦Abort⟧) or $t_{out}$ = $t_1$)
                      }
```

Figure 2.4: Type analysis of lambda calculus expressions using rewrite strategies

Chapter 3

Overview of Transformation Language *TL*

Type systems, which were discussed in the previous chapter, are typically oriented towards detection of run-time errors of a *particular* programming language. Therefore, before being able to concretely discuss the type system for program transformations, we need to review the key abstractions of a program transformation language, which are typically represented by:

Patterns that specify the object of interest in a transformational task. Patterns can be denoted in the concrete syntax of the object language or its abstract representation as terms. Patterns may contain meta- or schema variables, which range over terms and all sub-terms that can be derived from them.

Rewrite rules that specify a basic rewrite step in terms of input and output patterns. Application of a rewrite rule to a term proceeds by matching the term with the input pattern and applying the resulting meta-variable bindings to the output pattern in case of a successful match or returning failure otherwise. Rewrite rules may have optional conditions consisting of additional matches and other constraints that need to succeed for a rule to succeed. Such rules are also known as conditional rewrite rules.

Combinators that compose rewrite rules to build larger, more complex and strategic rewrites. The set of standard combinators includes sequential and conditional combinators. The sequential combinator requires both of its operands to succeed, while the conditional combinator allows a programmer to deal with an application failure strategically such that if the first operand fails, the second operand is attempted on an input term.

Iterators that from an operational perspective extend the application of a rewrite rule to a sequence of terms by enabling recursive invocation of a rule on a term and its sub-terms. Typical iterators are fixed point operators that exhaustively apply a strategy to the input term to compute its normal form and traversals that descend into the term's structure and apply a strategy to its sub-terms.

TL is a representative program transformation language in that it supports all of these transformational abstractions. Figure 3.1 summarizes the syntax of the language. In addition to the standard constructs, *TL* provides (1) higher-order rewrite rules, (2) transient strategies that are self-modifying during iterative applications, (3) a suite of operators for observing a strategy's application and a strategy's self-reduction, and (4) access to functional programming in Standard ML. The focus of the present discussion will be on standard transformational abstractions, but the non-standard features will also be discussed for a greater insight into the scope of the type analysis problem. The chapter concludes with an overview of related program transformation languages.

3.1 Patterns

Transformation languages are typically parametric with respect to the object language, whose terms are manipulated by a transformation program. The standard

p ::=		*patterns:*	s ::=		*strategies:*
	t	*terminal symbols*		p	*pattern*
	t_x	*schema variable*		r	*rewrite rule*
	$t[\![p^+]\!]$	*parse expression*		s t	*strategy application*
m ::=		*matching:*		uc s	*unary composition*
	true | false	*match constants*		s ⊕ s	*binary composition*
	p = s	*match equation*		fold ⊕ s	*higher-order comp.*
	m andalso m	*conjunction*	⊕ ::=		*binary combinators:*
	m orelse m	*disjunction*		<; | ;> | <* | *>	*sequencing comb.*
	not m	*negation*		<+ | +> | <+>	*choice-based comb.*
	sml.id(p^*)	*function calls*	uc ::=		*unary combinators:*
r ::=		*rewrite rules:*		FIX	*fixed-point iterator*
	p → s [if {m}]	*rule*		mapL | mapR | mapB	*one-layer traversals*
	ID	*identity*		hide | lift	*strategic application*
	SKIP	*no-op*		transient | opaque | raise	*strategic reduction*
d ::=		*declarations:*	prg ::=		*TL programs:*
	id: s	*rewrite abstraction*		d^+	*declarations*
	def id^+ = s	*iterator abstraction*			

Figure 3.1: *TL*'s syntax

means of specifying the syntax of an object language is a Context-Free Grammar (CFG) [28] in (Extended-) Backus-Naur Form (BNF) or abstract syntax notation [1][3]. The grammar with corresponding lexing and pretty-printing rules enables automatic construction of parsers and pretty-printers to convert object (language's) programs from text to trees and vice versa.

TL accepts CFGs in Extended-BNF (EBNF) notation and allows a programmer to use ML-Lex specification syntax for denoting lexing rules [4]. *TL* manipulates parse trees, which enables the preservation of as much of the input term's structure as possible. Pretty-printing rules use BNF grammar productions to specify proper formatting of parse trees. Additional details about the parsing and pretty-printing specifications together with examples can be found in [76].

Patterns in *TL* consist of terminal symbols, schema variables and parse expressions. Schema variables, denoted by t_x with an alphanumeric subscript x, range over the domain of all legal phrases α that can be derived from non-terminal symbol t: i.e. reflexive transitive closure of grammar derivations $t \overset{*}{\Rightarrow} \alpha$. A parse expression, denoted by $t[\![\alpha']\!]$, specifies a phrase in the concrete syntax of the object language; it is an instance of the transitive closure of grammar derivation $t \overset{+}{\Rightarrow} \alpha'$, where α' consists of terminal symbols and schema variables.

```
%LEFT_ASSOC "+" "L1"                                                  .
%LEFT_ASSOC "*" "L2"                                                  .

expr_list   ::=   expr [ expr_list ]                                  .
expr        ::=   expr "+" expr                         %PREC "L1"    .
expr        ::=   expr "*" expr                         %PREC "L2"    .
expr        ::=   boolean | integer | real | string | id | "(" expr ")"   .
id          ::=   idLex                                               .
boolean     ::=   boolLex                                             .
integer     ::=   intLex                                              .
real        ::=   realLex                                             .
string      ::=   stringLex                                           .
```

Figure 3.2: Language *Expr*

Sample EBNF grammar To observe specification of *TL* patterns consider the language defined by the EBNF grammar in Figure 3.2. Operator %LEFT_ASSOC specifies left-associative tokens. Tokens defined earlier in the lexical order have lower precedence than tokens defined later. Productions that use operator %PREC use the precedence of the operator's argument. Thus, because production `expr ::= expr "*" expr` is annotated with the precedence of `L2`, this production has a higher precedence level than all productions annotated with `L1`. If there is a choice between two or more productions in the expansion of a non-terminal symbol, the productions with the lowest precedence are chosen first. This allows expressions like `3 * 4 + 5 + 6` to be parsed as `((3 * 4) + 5) + 6` as opposed to `3 * ((4 + 5) + 6)` or `3 * (4 + (5 + 6))`. Symbols that are not defined by the grammar are lexer-level syntactic categories (e.g. idLex), literal values are surrounded by string delimiters (e.g. "("), and optional symbols are delimited by square brackets (e.g. [expr_list]).

Example The following are some examples of specifying patterns in the concrete syntax of the object language *Expr*:

- `expr`$_1$ – any term with the root `expr`

- `expr`⟦`x + y`⟧ – a term with root `expr` and leaves `x`, `+`, and `y`

- $\mathtt{expr}[\![\mathtt{x\ +\ id}_1]\!]$ – a term with a sub-term rooted in `id`

- $\mathtt{expr}[\![\mathtt{expr}_1\ \mathtt{+}\ \mathtt{expr}_1]\!]$ – a term with identical sub-terms.

□

3.2 Matching

A pattern is a formalism for specifying a set of terms. Membership in this set is determined by a matching algorithm, which can optionally support various equational theories such as associativity, commutativity and others [6][14]. *TL* uses syntactic matching (i.e. with the empty equational theory). Matching is performed by a *match equation* $t_l = t_r$, where t_r is a ground term (a term where all variables are bound to concrete values) and t_l is a pattern that may contain free variables. If a substitution σ can be constructed such that both sides of an equation are syntactically identical $\sigma(t_l) \equiv t_r$, then a match equation evaluates to $\langle true, \sigma\rangle$, and $\langle false, \varnothing\rangle$ otherwise.

A *TL match expression* is a boolean composition of match equations connected using *andalso*, *orelse*, *not*. A match expression succeeds producing $\langle true, \sigma\rangle$ if all constituent equations succeed using the standard boolean semantics and evaluation rules in Figure 3.3. To express the evaluation, we use the standard inference rule notation and evaluation semantics notation $expr \Downarrow \langle value, bindings\rangle$.

$\dfrac{\exists\sigma.\sigma(t_l) \equiv t_r}{t_l = t_r \Downarrow \langle true, \sigma\rangle}$	(MatchEq$^+$)	$\dfrac{e_1 \Downarrow \langle v_1, \sigma\rangle \quad \sigma(e_2) \Downarrow \langle v_2, \sigma'\rangle}{e_1\ andalso\ e_2 \Downarrow \langle v_1 \wedge v_2, \sigma'\rangle}$	(MatchConj)
$\dfrac{\nexists\sigma.\sigma(t_l) \equiv t_r}{t_l = t_r \Downarrow \langle false, \varnothing\rangle}$	(MatchEq$^-$)	$\dfrac{e_1 \Downarrow \langle v_1, \sigma\rangle \quad \sigma(e_2) \Downarrow \langle v_2, \sigma'\rangle}{e_1\ orelse\ e_2 \Downarrow \langle v_1 \vee v_2, \sigma'\rangle}$	(MatchDisj)
		$\dfrac{e \Downarrow \langle v, \sigma\rangle}{not\ e \Downarrow \langle \neg v, \sigma\rangle}$	(MatchNeg)

Figure 3.3: Matching

Example Match expression

$$expr_1 = expr[\![1+1]\!] \textit{ andalso } expr_2 = expr[\![2]\!]$$

evaluates to `true` based on a substitution

$$\sigma = \{(expr_1 \mapsto expr[\![1+1]\!]), (expr_2 \mapsto expr[\![2]\!])\}$$

The boolean-valued match expression $not(expr[\![1+1]\!] = expr[\![2]\!])$ also succeeds because the inner match equation fails with an empty substitution. □

3.3 Rewrite rules

Transformation is accomplished through the application of rewrite rules to terms. A rewrite rule

$$r: \; p \rightarrow e \; [\mathit{if} \, \{ \; m \; \}]$$

is a (labeled) abstraction with a pattern (or rule premise) p on the left-hand side, a strategy expression (or rule body) e on the right-hand side that may include other rules in case of higher-order rules, and match expression m in the optional condition. A rule r can be applied to a term t, denoted by $r\ t$, to rewrite it into e' if the shape of the argument term t matches the rule's premise p and match expression m succeeds, in which case the resulting substitution is applied to the right-hand side to produce e'. If the pattern match(es) fail, then rule application fails. Application failure is manifested differently in transformation languages classifying them into *identity-based* and *failure-based* languages. In an identity-based language, application failure leaves the input term unchanged, but propagates the boolean-valued failure to the enclosing context. In a failure-based language, application failure produces a distinguished meta-term (e.g. '↑') or throws an exception, which is observed and handled by the enclosing context. *TL* uses identity-based semantics [79].

Rewrite rules represent *scoped* contexts. An occurrence of term variable t_x in pattern p or match expression m is *bound* when it occurs in the body of a rule e. An occurrence of t_x is *free* if it appears in a position, where it is not bound by an enclosing rewrite rule. An expression with no free variables is said to be *closed.* In typical languages including *TL*, a valid rewrite rule is closed.

Two rewrite abstractions are built-in as primitives. Abstraction *SKIP* is a no-op that never applies to a term. Abstraction *ID* is an identity function that always applies to a term, but leaves it unchanged.

Example Application of rule

$$expr_1 \rightarrow expr_2 \ \mathit{if}\ \{expr[\![expr_2 + 0]\!] = expr_1\}$$

to $expr[\![x + 0]\!]$ succeeds with the substitution

$$\sigma = \{(expr_1 \mapsto expr[\![x + 0]\!]), (expr_2 \mapsto expr[\![x]\!])\}$$

and produces output term $expr[\![x]\!]$. On the other hand, if the rule is applied to term $expr[\![x + 1]\!]$, then the match expression fails leading to application failure, which in *TL* would leave the input term unchanged. □

3.4 Combinators

Rewrite rules can be composed using combinators to build strategies and more complex transforms. The set of binary combinators in *TL* consists of left-biased non-strict sequence $<\!;$, strict sequence $<\!*$, and choice $<\!+$, along with the right-biased counterparts $;\!>$, $*\!>$, $+\!>$, and non-deterministic choice $<\!+\!>$. The bias indicates which operand is applied to an input term first.

Composition $r_1 \mathbin{<\!;} r_2$ is a strategy that, when invoked on a term t, applies r_2 to the result of $(r_1\ t)$. To observe the output term in all four possible applications of constituent rules to the input term, suppose s, s_1 and s_2 are rules that always succeed and produce t_s, t_{s_1}, t_{s_2} respectively, and f is a rule that always fails and leaves the term unchanged. Then, the behavior of the left non-strict sequence is defined by the following:

$$(s_1 \mathbin{<\!;} s_2)\ t = s_2\ t_{s_1} = t_{s_2} \qquad (f \mathbin{<\!;} f)\ t = t \qquad (s \mathbin{<\!;} f)\ t = (f \mathbin{<\!;} s)\ t = s\ t = t_s$$

Alternatively, we can choose abstract strategies s, s_1, s_2 to be the concrete strategic constant *ID* and f to be the concrete strategic constant *SKIP*. In this case, the output term of both *ID* t and *SKIP* t is t, which makes t the result of all three equations.

Strict sequential composition $r_1 \mathbin{<\!\!*} r_2$ computes $(r_2\ (r_1\ t))$ only if both applications succeed, and leaves the input term unchanged otherwise: in other words

$$(s_1 \mathbin{<\!\!*} s_2)\ t = s_2\ t_{s_1} = t_{s_2} \qquad (f \mathbin{<\!\!*} s)\ t = (s \mathbin{<\!\!*} f)\ t = (f \mathbin{<\!\!*} f)\ t = t$$

Conditional composition $r_1 \mathbin{<\!\!+} r_2$ returns $(r_1\ t)$ if the first application succeeds, and $(r_2\ t)$ otherwise:

$$(s_1 \mathbin{<\!\!+} s_2)\ t = (s_1 \mathbin{<\!\!+} f)\ t = s_1\ t = t_{s_1} \qquad (f \mathbin{<\!\!+} s)\ t = s\ t = t_s \qquad (f \mathbin{<\!\!+} f)\ t = t$$

Finally, composition $r_1 \mathbin{<\!\!+\!\!>} r_2$ applies either of its operands to a term non-deterministically. Behavior of right-biased combinators is similar to the above with the difference of first attempting the right operand on the input term.

Example The following composition encodes the axioms of distributivity of multiplication over addition

$$\begin{array}{rcl} expr[\![expr_0 * (expr_1 + expr_2)]\!] & \rightarrow & expr[\![expr_0 * (expr_1) + expr_0 * (expr_2)]\!] \\ & \lhd\!\!\!+\!\!\!\rhd & \\ expr[\![(expr_0 + expr_1) * expr_2]\!] & \rightarrow & expr[\![(expr_0) * expr_2 + (expr_1) * expr_2]\!] \end{array}$$

Here, the composition encodes simplification for both cases of the multiplicand occurring to the left and to the right of a sum. In either case, it is distributed to both operands of the sum. The sum's operands are parenthesized so that if the operands are also sums, then the distribution can be applied again while preserving the meaning of the original input term. □

3.5 Iterators

Iterative application of a strategy to terms and sub-terms is controlled by iterators. Exhaustive application of a strategy to a term to compute the term's normal form can be performed using iterator *FIX*. The expression

$$\mathit{FIX}\ s$$

repetitively applies argument strategy s to an input term until successive applications no longer produce a syntactically distinct term.

To apply a strategy to sub-terms, *TL* provides one-layer, generic, primitive iterator *mapL*, which when combined with a strategy

$$\mathit{mapL}\ \ s$$

applies its argument strategy to all immediate sub-terms of a term from left to right. In other words, given term $t = f(t_1, t_2, \ldots, t_n)$, $(mapL\ s)\ t$ produces $f(t'_1, t'_2, \ldots, t'_n)$. If the input term does not have immediate sub-terms, then the iterators leaves the term unchanged.

Combinators, iterators and recursion can be combined to build full term iterators. For example, iterator definition

$$def\ TDL\ s = s{<\!;}mapL(TDL\ s)$$

defines a traversal scheme *TDL* that applies its argument strategy s to the root of a term and recursively applies itself to all immediate sub-terms. Recursion stops when no further sub-terms are found. This enables rewriting of all sub-terms within a term. Different iterators can be defined by re-arranging the order of application or by using a different combinator. For example, bottom-up traversal scheme can be defined with the following declaration

$$def\ BUL\ s = mapL(BUL\ s){<\!;}s$$

where application of argument strategy s to a term is preceded by a recursive descent into sub-terms. If success or failure of argument strategy in the application to sub-terms is important, then the definition can employ conditional combinators. For example, the following traversal scheme stops recursion into sub-terms after the first successful application to a sub-term:

$$def\ first\_TDL\ s = s{<\!\!\!+}mapL(first\_TDL\ s)$$

Example The following strategy will remove all additions with 0 by iterative top-down traversal of a term until no further such additions remain in the term:

$$\mathit{FIX}\ (\mathit{TDL}\ (\mathit{expr}[\![\mathit{expr}_x + 0]\!] \rightarrow \mathit{expr}_x))$$

□

3.6 Non-standard strategic controls

In addition to the standard strategic controls outlined above, *TL* provides a collection of features to allow programmers greater flexibility in programming transformations. This includes higher-order rules [83], transient strategies, operators for shielding the observability of application and reduction of strategies, and support for functional programming. We discuss these features next.

3.6.1 Higher-order rules and operators

A higher-order rule is a rule that can produce a new rule upon successful application to a term:

$$p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow p_n$$

Since higher-order rules may produce new higher-order rules, it is useful to distinguish them based on their order[65], which is inductively defined as follows:

- pattern p is a strategy of order 0 denoted by s^0;
- $s^0 \rightarrow s^n$ is a strategy of order $n + 1$.

One of the primary benefits of higher-order rules is to capture contextual information in a rewrite[79][11][72][51]. Contextual information is often needed because rewrite rules only have access to the term being rewritten, yet many transformation tasks require access to terms elsewhere in a program's tree: e.g. ancestor term higher-up in the tree or sibling term to a common ancestor term. Conceptually, if a rewrite of a term of interest requires the knowledge of term a, then the rewrite can be lifted to a higher-order rule, whose outer premise represents the term a. Then, the higher-order rule

$$p_a \rightarrow p_b \rightarrow p_c$$

can applied to term a in one part of a tree to produce rule $p_b \rightarrow p_c$, which has access to p_a bound by the application and which can then be used to perform the rewrite of interest in another part of a tree.

Compositions of higher-order rules need to account for composition of new dynamic rules generated by application. This is accomplished by operator *fold*, which folds newly generated dynamic rules into a dynamic strategy using its argument combinator:

$$\mathit{fold} \; \oplus s$$

Example The following strategy extracts type information from declaration statements of an object language and (with proper traversals) rewrites references to the declared identifiers into their types (e.g. for further type-checking):

$$\begin{array}{ll} \mathit{fold} \; <\!; & (\\ & \quad \mathit{stmt}[\![\mathit{int}\ id_x;]\!] \rightarrow id_x \rightarrow id[\![\mathit{intTy}]\!]) \\ & \quad <\!\!+\!\!> \\ & \quad \mathit{stmt}[\![\mathit{double}\ id_y;]\!] \rightarrow id_y \rightarrow id[\![\mathit{doubleTy}]\!] \\ &) \end{array}$$

Here, upon encounter of a term rooted in *stmt* a new dynamic rule will be generated if the shape of the input term fits one of the premises of the two higher-order rules. If two dynamic rules are generated during a traversal, then the rules will be composed using non-strict sequential combinator $<;$. □

To iterate over sub-terms with higher-order rules, we can define higher-order full-term traversals. For example, the following declaration defines a top-down traversal that composes dynamic rules using the choice combinator:

$$def\ \ lcond\_tdl\ s = fold{<\!\!+}(s{<;}mapL(lcond\_tdl\ s))$$

In other words, the main difference between first-order and higher-order traversal definitions is the addition of operator *fold*.

Example The following strategy collects dynamic rules in the first pass through a program and rewrites identifiers into their types in the second pass:

$prog_{in} \rightarrow BUL\ (\ lcond\_tdl\ ($

$\qquad\qquad stmt\llbracket int\ id_1\rrbracket \rightarrow id_1 \rightarrow id\llbracket intTy\rrbracket{<\!\!+\!\!>}$

$\qquad\qquad stmt\llbracket double\ id_2\rrbracket \rightarrow id_2 \rightarrow id\llbracket doubleTy\rrbracket$

$\qquad\qquad)\ \mathrm{prog}_{in}$

$\qquad)\ \mathrm{prog}_{in}$

□

3.6.2 Transient strategies

A transient strategy is a self-modifying strategy that reduces to no-op *SKIP* after the first successful application:

$$transient\ s$$

Transient strategies enable specification of higher-order rules that generate dynamic rules that can apply only once. This behavior is useful in a variety of transformation tasks, where only one successful rewrite is required [79]. In addition, combinator *transient* together with one-layer iterator *mapL* can be used to choose to apply a strategy only to the first sub-term:

$$mapL(transient\ s)$$

where after the first successful application, strategy s reduces to no-op *SKIP* , which will leave remaining sub-terms unchanged. This is a generalization of iterator *one* in transformation system Stratego [69] and related systems.

Since strategies are reducible in iterative applications, *TL* extends the behavior of one-layer iterator *mapL* with iterators *mapR* and *mapB*. One-layer iterator *mapR* applies its arguments strategy to sub-terms of an input term in the *right-to-left* order as opposed to the left-to-right order used by *mapL*. This behavior can be used to choose to apply a strategy only to the last sub-term of an input term:

$$mapR(transient\ s)$$

One-layer iterator *mapB* iterates over sub-terms using a *copy* of its argument strategy. This allows a programmer to ensure that reduction of a strategy applied to one sub-term is not observable in application to other sub-terms. Thus, if r is a rewrite rule, the following equalities demonstrate the behavioral difference of *mapB* from other one-layer iterators:

$$mapB(transient\ r) = mapL\ r = mapR\ r$$

which in other words states that application of the three expressions to an input term will produce identical output terms.

For additional flexibility in controlling the conditions under which a (dynamic) strategy reduces to *SKIP* , *TL* provides combinators *opaque* and *raise*. Combinator *opaque* hides successful application of its argument strategy from a lexically enclosing *transient*. In other words, if s is a strategy that always succeeds, then iterative application of the following strategy will not reduce to *SKIP* :

$$\mathit{transient}(\mathit{opaque}\ s)$$

On the other hand, combinator *raise* enables propagation of a successful application beyond a lexically enclosing *opaque* and application of the following strategy to a term will always reduce the strategy to *SKIP* :

$$\mathit{transient}(\mathit{opaque}(\mathit{raise}\ s))$$

TL provides two additional combinators that modify the application, instead of a reduction, of a (dynamic) strategy. Combinator *hide* prevents the lexically enclosing combinators from observing the successful application of its argument strategy. If s is a strategy that always succeeds, then the following equality holds

$$(\mathit{hide}\ s) \mathbin{\lhd\!\!\!+} s = s \mathbin{<\!;} s$$

Combinator *lift* propagates successful application of its argument strategy beyond a lexically enclosing *hide*. If s always succeeds and f always fails, then

$$\mathit{hide}(\mathit{lift}\ \mathit{ID}) \mathbin{\lhd\!\!\!+} s = \mathit{ID} \qquad \mathit{hide}(\mathit{lift}\ f) \mathbin{\lhd\!\!\!+} s = s$$

For further details and examples on transient strategies and corresponding combinators see [79] and [81].

3.6.3 Functional programming

TL provides access to functional programming by allowing a programmer to invoke functions written in Standard ML (SML) from within a rewrite rule condition's body:

$$sml.id()$$

For example, given a function *output* that prints its argument to the console, the following rule

$$expr[\![expr_1 + 0]\!] \rightarrow expr_1 \ if \ \{sml.output(\text{``}simplified\ an\ addition\ with\ zero\text{''})\}$$

will log a console statement upon successful application of the rule.

Access to SML functions provides further flexibility by enabling access to all capabilities of a general-purpose language, which includes interaction with the operating system's environment and state. For details on the specification of SML functions for access by *TL* programs see [76].

3.7 *TL* programs

A *TL* program consists of a list of strategy declarations: both non-recursive abstractions of the form $id : s$, where id serves as a shorthand identifier for strategy s, and recursive abstractions of the form $def\, id\ args = s$, where strategy s may refer to formal arguments $args$ and id.

A distinguished *main* rule must be defined to drive transformation by rewriting a term rooted in the start symbol of the object language's grammar. Elaboration of a *TL* program is initiated by applying the main rewrite rule, which may invoke other strategies on the input term. If the result of elaboration is not a term or if an error is raised during elaboration, the computation terminates abnormally. Otherwise, the result is a new term in case of successful application of *main* or else it is the unchanged input term. Figure 3.1 summarizes the syntactic constructs of *TL* discussed above.

3.8 Related work

Besides *TL*, there are a number of other program transformation systems. Some examples are ELAN, ASF+SDF, Stratego, Strafunski, DMS, and CT. We review each of them below and discuss their error detection and analysis capabilities.

ELAN [10][58] is a system that provides a first-order rewrite specification language that is similar to the algebraic specification formalisms of traditional rewrite systems such as Maude [15]. The syntax of the object/term language is defined through modularized sort definitions. Rewrite rules follow the form:

```
⟨rule⟩        ::=  "[" ⟨label⟩? "]" ⟨term⟩ "=>" ⟨term⟩ ⟨localeval⟩*
⟨localeval⟩   ::=  "if" ⟨boolean_term⟩
                |  "where" ⟨variable⟩ ":=" "(" ⟨strategy⟩? ")" ⟨term⟩
                |  "where" ⟨term⟩ ":=" "(" ⟨strategy⟩? ")" ⟨term⟩
                |  "choose" ("try" ⟨localeval⟩+)+ "end"
```

where "quoted" symbols denote terminal tokens, parentheses are used to group symbols, * denotes zero or more repetitions, $^+$ denotes one or more repetitions, and ? denotes zero or one occurrence of a symbol.

Unlabeled rewrite rules are used by the system to reduce the input term to a normal form using a fixed "leftmost-innermost" evaluation strategy (i.e. a *BUL* traversal in *TL*). Labeled rewrite rules can be invoked by a user to be applied to a term that has already been normalized by unlabeled rewrite rules. Similar to *TL*, rules can have conditions or local evaluations that capture

1. evaluations yielding a boolean outcome,
2. local assignment, where a local variable can store intermediate results of strategy application,
3. local matching, where variables within a term can be matched against results of strategy application or
4. composition of local evaluations, which chooses the first successful evaluation.

ELAN supports several equational matching theories such as associative and commutative theories, which it uses to enumerate all matching results. Application of a strategy to a term produces a set of terms. This, coupled with a backtracking capability, allows this system to enumerate all possible results similar to Prolog. Unlike *TL*, failure of a strategy to apply to a term is indicated by an empty result set. In addition to conditional rewrite rules, ELAN provides the following combinators, which can compose rules to form strategies:

1. sequence $s_1; s_2$, which is similar to $s_1 \mathbin{<\!\!*} s_2$ in *TL* ,
2. non-strict sequence (*dont know choose*) $dk(s_1, \ldots, s_n)$, which is similar to $s_1 \mathbin{<\!;} \ldots \mathbin{<\!;} s_n$ in *TL* ,
3. non-deterministic choice (*dont care choose*) $dc(s_1, \ldots, s_n)$, which is equivalent to $s_1 \mathbin{<\!\!+\!\!>} \ldots \mathbin{<\!\!+\!\!>} s_n$ in *TL* ,
4. left-biased choice *first*$(s_1, \ldots, s_n)$, which is equivalent to $s_1 \mathbin{<\!\!+} \ldots \mathbin{<\!\!+} s_n$ in *TL* ,
5. singleton selector *one* that returns a non-deterministically chosen element from a set of results,
6. identity rewrite *id*, which is equivalent to *ID* in *TL* ,
7. failing rewrite *fail* that always fails,

8. iterators *repeat*$^*(s)$ and *iterate*$^*(s)$, which are similar to $FIX(s)$ in TL .

Unlike other transformation systems, ELAN does not support generic traversals [40].

Analysis and error detection capabilities in ELAN are limited to well-formed syntax checks performed by the ELAN program parser. We are not aware of a static (type) system to detect errors in strategies.

ASF+SDF [67] is a transformation system that uses the Syntax Definition Formalism (SDF) to specify object/term languages and the Algebraic Specification Formalism (ASF) to specify rewrite rules. Rewrite rules follow the form:

```
⟨rule⟩        ::=   "[" ⟨label⟩ "]" ⟨term⟩ "=" ⟨term⟩ ("when" ⟨condition⟩)?
⟨condition⟩   ::=   ⟨term⟩ ":=" ⟨term⟩
                |   ⟨term⟩ "!:=" ⟨term⟩
                |   ⟨term⟩ "==" ⟨term⟩
                |   ⟨term⟩ "!=" ⟨term⟩
```

The optional conditional part of rewrite rules allows match (:=) and non-match (!:=) conditions that perform syntactic first-order matching as in TL and equality (==) and inequality (!=) conditions that perform syntactic identity checks. In either case, terms are reduced to normal forms before matching and equality conditions are evaluated. The textual order of rewrite rules in an ASF module defines the order of rules in the sequential composition. The default evaluation strategy is "leftmost-innermost" similar to ELAN. The input term is normalized using rewrite rules until no further reductions are possible. However, the system does not check for confluence and termination, which implies that the user must perform these checks. In ASF+SDF, failure of a rule to apply to a term is handled similar to TL such that the resulting term of application failure is the input term. In addition to the default evaluation strategy, programmers can define new traversals using three constructs of *transformers*, *accumulators* and *accumulating transformers*. Transformer traversals modify a term directly, whereas accumulators collect information about a term during traversal of the term. Accumulating transformer combines the transformation and accumulation. New traversals need to account for each term signature that is involved in transformation or accumulation. Therefore, new and custom traversals are not generic and may involve a substantial programming effort in a setting where many term signatures need to be accounted for.

Similar to ELAN, error-detection capabilities in ASF+SDF are limited to parser-enforced well-formed syntax checks.

Stratego [69][11] uses the SDF notation to specify the syntax of object/term languages. However, unlike ASF+SDF, it provides a richer set of strategic control constructs. In particular, Stratego provides lower-level primitive operations to match a pattern ($?p$) and to use the resulting substitution to build a ground term from a pattern ($!p$). Thus, rewrite rules can be defined in terms of the sequential composition of these primitives: $?lhs; !rhs$. The syntax of rewrite rules follows the form

⟨rule⟩ ::= ⟨label⟩ ":" ⟨term⟩ "->" ⟨term⟩ ("where" ⟨strategy⟩)?

The "where" clause can contain any strategy and the outcome of the clause is a boolean value to indicate success or failure of the strategy. There is no implicit evaluations like in ASF+SDF or ELAN and the evaluation is entirely user-controlled as in *TL*. Sequential ";" and choice " $<\!+$ " combinators can be used to control strategy composition. The built-in strategy *id* behaves similar to *ID* in *TL*; $fail$ is a strategy that always fails. In Stratego, application failure produces a distinguished meta-term $\uparrow$, which can be handled by an enclosing choice combinator. To avoid ill-formed terms during application, a primitive *try* is provided to leave the input term, on which a strategy failed, unchanged: i.e. $try(s) <\!+ id$. Generic traversal operators *all* and *one* can be used to build term traversals to apply an argument strategy to either all sub-terms or the first sub-term from the left. Thus, the suite of strategic control combinators and generic traversals provide very flexible means of strategic programming.

We are not aware of static analysis tools for Stratego that would assist programmers in static error detection.

DMS [9][2] is a commercial, software re-engineering toolkit maintained by Semantic Designs, Inc. DMS is implemented in PARLANSE (PARallel LANguage for Symbolic Execution) [8] that is a C-based rewriting language. DMS has been used in program transformations of over 20 languages ranging from general-purpose languages like C, C++ and Java[26][47][29] to domain-specific languages like SQL. A Unicode-supporting lexer with a GLR-based parser create efficient abstract syntax trees (AST), stored internally as hyper-graphs. The ASTs are subjected to three kinds of rewriting operations:

1. encoding of semantic actions (procedures) within a grammar in the style of YACC grammar production actions,
2. analysis of programs (analyzers) using attribute grammars, where a node has several attributes providing additional information that can be either inherited or synthesized and stored in a symbol table,
3. rewriting of terms (transforms) in DMS's Rule Specification Language (RSL), which supports Associative-Commutative matching.

Rewrite rules follow the form:

```
⟨rule⟩   ::=   “rule” ⟨id⟩ “(”⟨id⟩“:” ⟨type⟩“)” “:” ⟨type⟩ “->” ⟨type⟩ “=”
               ⟨term⟩ “rewrites to” ⟨term⟩ [“if” ⟨predicate⟩]
```

Transformation results can be pretty-printed in either a *default* mode, where formatting rules, specified in the object grammar, are used to produce appropriate spacing, or *fidelity* mode that reproduces original text verbatim. Generic traversals within

DMS are implemented using bi-directional edges that connect neighboring nodes in an AST's hyper-graph. While there are no pre-defined abstractions of strategic combinators, the control over rewriting is expressed by imperative procedures. Studies show that DMS has taken 50 person-years to develop and has been applied to transformation problems with over 4000 files and over 2.5 MSLOC (millions of source lines of code) [2].

We are not aware of any static analysis tools to detect errors in program transformations written in PARLANSE beyond C-based static analysis tools such as `lint`.

CT [37][5] is a logic-based *C*onditional *T*ransformation system. Terms of an object language are represented by a propositional fact base. For example, the term *class(2, 1, 'MyClass')* represents a *class* node in the Abstract Syntax Tree of an object program with identifier 2, parent identifier 1 and label *MyClass*. The parent-child relationships that are implicit in parse trees or typical AST's are represented by explicit references to node identifiers. Terms are represented by a logic predicate *exists* that is quantified over all facts. So, the node above is represented in logic as *exists(class(2, 1, 'MyClass'))*. Having represented a program as a fact base, a programmer can build patterns using logic variables: e.g. *exists(class(X, Y, 'MyClass'))*. Rewriting is accomplished by predicates *add(fact)*, *remove(fact)*, and their sequential composition. Rewrite rules in this framework follow the form similar to the following:

```
⟨rule⟩        ::=  ⟨id⟩ "=" ⟨cond⟩ "|" ⟨instance⟩ "->" ⟨transform⟩
⟨cond⟩        ::=  "exists(" ⟨fact⟩ ")" | ⟨cond⟩ "and" ⟨cond⟩
                |  ⟨cond⟩ "or" ⟨cond⟩ | "not" ⟨cond⟩
⟨instance⟩    ::=  "new_id(" ⟨id⟩ ")" | ⟨instance⟩ "," ⟨instance⟩
⟨transform⟩   ::=  "add(" ⟨fact⟩ ")" | "remove(" ⟨fact⟩ ")"
                |  ⟨transform⟩ "," ⟨transform⟩
```

Operationally, evaluation proceeds by a pattern match in a rule's condition, instantiation of all remaining free variables by the $new\_id(X)$ function and a global state change by adding or removing facts into the initial fact base. Individual rewrite rules can be combined into larger transformations with AND- and OR-sequences using "and-seq(rule, rules)" and "or-seq(rule, rules)".

Due to the logic-based framework, CT represents a non-traditional approach to rewriting [54] [53] [39]. The primary advantage of this approach is that traversals are not needed because all terms are available in a flat fact base. However, it appears that the responsibility for maintaining integrity of parent-child links between nodes is delegated to the programmer instead of being enforced by a parser as in other systems.

We are not aware of any static analysis tools to detect errors in program transformations within CT.

Strafunski [44][43] is different from the other transformation systems in that instead of providing a dedicated specification language for encoding of strategies it leverages generic functional programming[45][38] and encodes strategies in the Haskell language. In particular, it uses Haskell functions to encode rewrite rules and provides a library of functions for strategic combinators and generic traversals. A typical transformation scenario in this framework consists of the following steps:

1. convert an object/term language grammar into an algebraic data type specification so that a term's parse tree can be viewed as an algebraic data structure,

2. define Haskell-based functional strategies,

3. apply the resulting program on the data structure, and

4. pretty-print the resulting data structure into term language text.

As part of a research program related to Strafunski, Lämmel and colleagues have identified the importance of error detection in strategic program as a major research problem [42]. For example, in [40], a somewhat conservative type system for rewrite strategies, including traversal strategies, was developed. More specifically, system S—the core of program transformation system Stratego [69]—is extended with new syntax and semantics to support types. Our contributions advance this work by improving the expressivity of type analysis. Most notably, instead of using *sorts* to assign types to rewrite rules, we incorporate *constructors* into types. This improves the analytical precision because application of a rule, which acts on one constructor of a sort, to a term derived from a different constructor of the same sort, while being well-typed under the previous approach, will always fail, which is statically detected under our approach.

An extended strategic rewriting core language that includes one-layer traversals over a single-sorted term language of natural numbers has been recently formalized in an Isabelle/HOL-based model [33]. There, success and failure behavior of strategies has been analyzed from the perspective of *infallibility*: i.e., does a given strategy always succeed? Our analysis takes a dual view on this issue from the perspective of successfulness: i.e., can a given strategy succeed? Both approaches approximate correctness: in the first case by flagging over-specified strategies, and in the second case by flagging under-specified strategies.

XSLT Among the related work on transformations of tree-structured data are the W3C standards on XPath, XQuery and XSLT [71]. In particular, an XSLT style-sheet uses XPath and/or XQuery expressions to select elements within an XML document and uses templates to transform the elements. In other words, selection criteria could be viewed as pattern matches and templates as rewrite rules among many other cross-domain similarities [41, 18]. In this domain, an important type-checking question is whether the result of an XSLT transformation conforms to an intended type. This is due to the asymmetry arising from parser-based validation on inputs, but none on outputs. To cope with this problem, regular expression types [28] have been proposed to validate XML transformations [30] along with efficient type-checking implementations [23]. Other related type-checking questions are whether selection criteria return an empty set of nodes leading to a template that can never fire and whether one template is subsumed by another.

Chapter 4

Type Analysis of Rewrite Strategies

In the two previous chapters, we reviewed type systems and the transformation language whose programs are the intended targets of type analysis. In this chapter we turn to the main contribution of this research – type analysis of term rewriting strategies.

We begin the discussion with the definitions of types T, the relation between strategies and their types $s : T$ and the typing environment Γ, which is used to store and retrieve type bindings. Next, we develop a type system that calculates types of strategies. The type system is presented in the form of typing rules for the syntactic constructs of the language: starting from the patterns and pattern matching up through strategies, strategy definitions and programs. Following the definition of typing rules, we provide concrete examples and their type-checked results to ground the analysis in action.

Note that the focus of this chapter is on the core features of the type system – analysis of rewrite rules and their strategic compositions. Analysis of iterators and non-standard features of the transformation language such as higher-order rules and compositions will be discussed in the following chapters as extensions of the core system.

4.1 Motivation

In order to set the stage for the discussion of the analysis, let us first compare the analysis performed by Standard ML's type system to the analysis needed for rewrite strategies.

The main distinction between functions and rewrite rules is that functions in ML are required to be total in that given $f : I \to O$ it is required that $\forall x \in I.\ f(x) \in O$ or in other words $\nexists x \in I.\ f(x) \notin O$. A function may be defined using patterns $p_i \in I$ such that each p_i is mapped to value $v_i \in O$. In contrast, rewrite rules are at a *lower* level of abstraction such that a rule is *single* mapping between input and output patterns and is inherently non-total. A function can be defined using a set of conditionally composed rewrites. Because of this abstraction shift, functions can be viewed as rewrites of domains, whereas rules are rewrites of terms within a domain. The primary implication is an *impedance mismatch* between problem-specific rewriting needs and language-enforced rewriting requirements.[1] In other words, if a problem requires rewriting of a few selected terms of interest, a functional language may not be the best choice.

As a concrete example, consider the task of simplifying arithmetic expressions defined by the following ML data type:

[1] Impedance mismatch refers to conceptual and technical difficulties encountered while working with tools that are not well-suited for a problem at hand. E.g. object-relational impedance mismatch occurs while reading/writing object-oriented data from/to relational databases.

```
datatype Arith  =  mult of Arith * Arith
                |  plus of Arith * Arith
                |  num of int
                |  var of Id
   datatype Id  =  X | Y | Z
```

To distribute multiplication over addition, we could define the following function

```
fun  distributeF(mult(a, plus(b,c))  = plus(mult(a,b), mult(a,c))
  |  distributeF(mult(plus(a,b),c))  = plus(mult(a,c), mult(b,c))
```

The same functionality encoded using strategies employs conditional composition of two identity-based rewrite rules:

distributeS:

$$\texttt{Arith}[\![\texttt{mult(Arith}_a\texttt{, plus(Arith}_b\texttt{,Arith}_c\texttt{))}]\!] \quad \rightarrow$$

$$\texttt{Arith}[\![\texttt{plus(mult(Arith}_a\texttt{, Arith}_b\texttt{), mult(Arith}_a\texttt{, Arith}_c\texttt{))}]\!]$$

$$\lhd\!\!+$$

$$\texttt{Arith}[\![\texttt{mult(plus(Arith}_a\texttt{,Arith}_b\texttt{), Arith}_c\texttt{)}]\!] \quad \rightarrow$$

$$\texttt{Arith}[\![\texttt{plus(mult(Arith}_a\texttt{,Arith}_c\texttt{), mult(Arith}_b\texttt{, Arith}_c\texttt{))}]\!]$$

Note that strategic combinator $\lhd\!\!+$ is similar to functional operator | with respect to the choice and ordering of pattern mappings.

Now, suppose we extend the algebraic simplification to include the multiplication of a sum with the absorbing element 0 defined by a new function:

```
fun  multZeroF(mult(num(0), plus(a,b)))  = num(0)
  |  multZeroF(mult(plus(a,b), num(0)))  = num(0)
```

and compose the new function with the existing function such that if neither function applies, then the input term is left unchanged:

```
fun compF t = distributeF t handle Match =>
```

```
                multZeroF t handle Match => t
fun fixedP t= let val t' = compF t
              in  if t = t' then t' else fixedP t'
              end
```

Note that the composition of functions is strategic with respect to application failure (in this case runtime exception `Match`) such that if the first function fails to apply, then the second one is attempted.

The same extension encoded using strategies defines a new strategy for the extension and conditionally composes it with the existing strategy `distributeS`:

`multZeroS:`

$$\texttt{Arith}\llbracket\texttt{mult(num(0), plus(Arith}_a\texttt{, Arith}_b\texttt{)}\rrbracket \rightarrow \texttt{Arith}\llbracket\texttt{num(0)}\rrbracket$$

$$\lhd\!\!+$$

$$\texttt{Arith}\llbracket\texttt{mult(plus(Arith}_a\texttt{, Arith}_b\texttt{), num(0)}\rrbracket \rightarrow \texttt{Arith}\llbracket\texttt{num(0)}\rrbracket$$

$\texttt{compS:}\quad \texttt{FIX(distributeS} \;\lhd\!\!+\; \texttt{multZeroS)}$

Having considered the similarities, let us now analyze the behavior of the composed functions. In particular, consider the following invocation:

```
fixedP(mult(num(0), plus(var(X), var(Y)))) =
      plus(mult(num(0),var(X)),mult(num(0),var(Y)))
```

In other words, the composition failed to simplify the term to `num(0)` as we would normally expect. The culprit is the incorrect strategic ordering of functions such that application of one function makes the other function unreachable: pattern `mult(a,...)` is more generic than the pattern `mult(num(0),...)`. This can be fixed by re-ordering of the functions:

```
fun compF' t = multZeroF t handle Match =>
                 distributeF t handle Match => t
```

Note that the (Standard) ML's type system does not perform sufficiently detailed analysis of strategic compositions (expressions of the form *expr* `handle` *p* `=>` *expr*) to detect the unreachable functions.

While it may be possible to combine the patterns of the two functions into a single function, which can then be analyzed by the SML's type system for the non-exhaustive and/or redundant pattern matches[65], the problem presented in this example does not go away. For example, conditional compositions of rewrite rules allow rules to be drawn from heterogeneous types, whereas all patterns of a function must come from a single type. Thus, if patterns of functions `distributeF` and `multZeroF` were to come from two distinct types, then the functions could not be combined to employ SML's analysis of patterns.

As another example, consider ML function composition operator $\cdot \circ \cdot$, whose counterpart in strategic rewriting is the sequential combinator $\cdot <\!\!* \ \cdot$. Suppose we were to compose the aforementioned functions sequentially. In other words:

```
val  seqF = multZeroF o distributeF
```

Note that ML's type system allows this, even if it is clear that the composition will always fail for all inputs. Even if the first function (`distributeF`) does apply, it produces a term that cannot be handled by the second function (`multZeroF`). On the other hand, strategic composition

```
seqS: distributeS <* multZeroS
```

will be flagged as a type error because none of the output patters of the first strategy fit the input patterns of the second.

The type system for strategies presented here goes much further than ML's type system in the analysis of compositions. It performs analysis at the level of individual rewrites, instead of (data) type rewrites, and allows patterns to have an arbitrarily deep and wide (recursive) structure. This enables detection of unreachable and other kinds of erroneous strategies (as summarized in Chapter 1).

4.2 Types, contexts and the typing relation

Terms of an object language in TL are tree-structured entities with leaves belonging to the domain of terminal symbols and interior nodes belonging to the domain of non-terminal symbols of the object language's grammar: in other words

$$\begin{aligned} t &= c(t_1, \ldots, t_n),\ n \geq 0 \\ c &= M \cup N \end{aligned}$$

where M denotes the set of terminal symbols and N is the set of non-terminal symbols. Thus, $integer(0)$ is a term with 0 as a sub-term, which does not have any of its own sub-terms.

The classic approach of assigning types to terms is to assign a term the type of its root symbol [40][33]. Using such an approach, the type of term $expr[\![true]\!]$ is $expr$ and the type of term $integer[\![3]\!]$ is $integer$, thus simplifying the type analysis by abstracting away from the types of sub-terms. The classic approach is useful for structurally simple values, but is overly conservative in a domain, where the grammar of a language can define more than one production for a non-terminal symbol. For example, symbol $expr$ in Figure 3.2 can derive substantially different strings making terms like $expr[\![true]\!]$ and $expr[\![3]\!]$ receive the same type $expr$ under this approach.

The implication of the classic approach is a substantial conservativeness in the resulting type analysis. The root symbol of a term on its own does not provide sufficient information about the kinds of sub-terms the term contains. For example, if all that a type system can tell about rewrite rule $expr[\![id_x + 0]\!] \to expr[\![x]\!]$ is that it is of type $expr \to expr$, then application of the rule to term $expr[\![true]\!]$ is valid from such typing perspective, when in fact we can statically determine that the application has no chance of succeeding at run-time. Further, a strategy may traverse over sub-terms of an input term. Presence or absence of certain kinds of sub-terms is a prerequisite information in determining whether application of a traversal strategy to a term is valid. So, if all that a type system can tell about an input term $expr[\![true]\!]$ is that it is of type $expr$, then application of a traversal strategy $TDL\ (id[\![x]\!] \rightarrow id[\![y]\!])$ to such term cannot be analyzed to determine if it has a chance of succeeding. This is because the most that a type system can determine from the input term's type $expr$ is that it can potentially derive a term of type id. However, it is possible to statically determine that term $expr[\![true]\!]$ does not contain a sub-term of type id. These kinds of type errors cannot be detected by the classic typing approach.

To enable detailed type analysis of rewrite strategies, we adopt a richer model of types, where a term's tree structure itself is the type of the term. Further, the type of a term depends not only on the structure of the term, but also on the leaves of a term making our types similar in spirit to dependent types, where types depend on arbitrary values[84]. Thus, taking the grammar G of term language $\mathcal{L}$ as a parameter such that $t \in \{N \cup M\}$, where N is the set of non-terminal symbols and M is the set of terminal symbols, a term's structure and its type can be generically represented by a product of a root symbol t and a list of sub-term types. Therefore, a *term type* is defined by the following

$$\mathrm{T ::= t \times T\ list}$$

Patterns may contain schema or term variables, which do not have a structure. Schema variables become bound at run-time during evaluation. To enable analysis of unbound structures, the type system supports type variables denoted by ‘x. Other kinds of type constructs are summarized in Figure 4.1.

A *typing context* (or environment) Γ is a set of bindings $t_x : T$ mapping term variables to their types. In addition, contexts track bound type variables so that whenever a fresh type variable is required it is chosen to be distinct from all previously bound type variables. Finally, since *TL* programs are lists of labeled abstractions, which may refer to other abstractions defined elsewhere in a program, we track bindings of abstraction labels to types $x : T$.

The classic typing relation $t : T$ connects terms to their types [60]. Due to the presence of variable bindings, the typing relation is extended to $\Gamma \vdash t : T$, where Γ is the typing context. Because our type system also needs to support type inferencing, analysis of a term may affect previous bindings. Therefore, we adopt the following *typing relation*

$$\Gamma \vdash e : \langle T, \Gamma_1 \rangle$$

which states that, under a typing context Γ, the type of expression e is T and the resulting typing context (due to type inferencing) is Γ_1.

Γ	::=		*contexts:*
		$\emptyset$	*empty context*
		Γ, t_x: T	*term variable binding*
		Γ, ‘x	*type variable binding*
		Γ, x : T	*abstraction variable binding*
T	::=		*types:*
		t $\times$ T list	*terms*
		‘x	*type variable*
		T $\rightarrow$ T	*rewrite rules*
		T + T	*choice*

Figure 4.1: Types and typing contexts

4.3 Typing of patterns

The types of patterns are determined by (possibly recursive) application of the typing judgements listed in Figure 4.2.

Rule `p-var` assigns term variable t_x the type stored in environment Γ. Rule `p-leaf` states that the type of a term with no sub-terms is $(t, [\ ])$ under any environment. Rule `p-tree` computes the type of individual sub-terms prior to assigning a type to

$$\frac{t_x : T \in \Gamma}{\Gamma \vdash t_x : \langle T, \Gamma \rangle} \quad \texttt{(p-var)}$$

$$\frac{}{\Gamma \vdash t : \langle (t, [\,]), \Gamma \rangle} \quad \texttt{(p-leaf)}$$

$$\frac{\forall i.\ \Gamma_{i-1} \vdash p_i : \langle T_i, \Gamma_i \rangle}{\Gamma_0 \vdash (t, [p_i^{i \in 1..n}]) : \langle (t, [T_i^{i \in 1..n}]), \Gamma_n \rangle} \quad \texttt{(p-tree)}$$

Figure 4.2: Typing of patterns

the entire term.

To emphasize structure and to express the complete details of a term, the inference rules use a deeply structured tuple notation for *terms* – $(t, [p_i^{i \in 1..n}])$, where p_i are immediate sub-terms, instead of the more readable parse expression notation – $t[\![p_i^{i \in 1..n}]\!]$, where p_i are tree leaves. When the structure of a term's type can be elided, we will similarly use a more readable representation of the deeply structured *type* – $(t, [T_i^{i \in 1..n}])$ as $t[T_i^{i \in 1..n}]$. Further, we use $T_i^{i \in 1..n}$ with $n \geq 1$ instead of $T_1, \ldots, T_n$ to avoid any notational complexities (see for example [60]).

Example Let us use the typing rules of Figure 4.2 to calculate types of some patterns of object language *Expr* defined in Figure 3.2:

1. type of terminal symbol $+$ is $+$ or with full structure $(+, [\,])$
2. type of parse expression $expr[\![x]\!]$ is $expr[x]$ or with full structure $(expr, [(id, [(idLex, [(x, [\ ]\ )]\ )]\ )]\ )$
3. type of parse expression $expr[\![0]\!]$ is $expr[0]$ or with full structure $(expr, [(integer, [(intLex, [(0, [\ ]\ )]\ )]\ )]\ )$
4. type of term variable $expr_1$ under context $\{expr_1 : expr[x + y]\}$ is $expr[x + y]$ or with full structure

$(expr, [(expr, [(id, [(idLex, [(x, [\ ]\ )]\ )]\ )]\ ), (+, [\ ]),$

$(expr, [(id, [(idLex, [(y, [\ ]\ )]\ )]\ )]\ )])$

4.4 Typing of matches

A match equation is a syntactic match of term patterns, where the right operand is a ground term and the left operand may contain unbound term variables:

- If the left operand is a ground term, then the match equation is used for syntactic comparison of two ground terms, which produces a boolean value and an unchanged substitution.

- If the left operand contains variables, then the match equation is used for binding term variable(s) for further use in the enclosing rewrite rule. In this case, any term variable within the left operand is *implicitly declared* and bound to a corresponding (sub)term of the right operand. The result of such match equation is a boolean value indicating structural compatibility of the two operands and a substitution that binds variables to terms.

For example, consider the following conditional rewrite rule

```
expr₁ → expr⟦id₁⟧
if { expr⟦x⟧ = expr₁ andalso
     expr⟦id₁⟧ = expr₁
}
```

Here, schema variable `expr`$_1$ is implicitly declared by the rule's left-hand side and becomes bound when the rule is applied to a matching term. The first match equation is a syntactic comparison equation because both operands are ground terms. The second match equation is a binding equation. In addition, schema variable `id`$_1$ is implicitly declared. Finally, the rule's right-hand side refers to the bound schema variable `id`$_1$.

To record term variables implicitly declared by match equations, we introduce a relation that updates an environment

$$\Gamma_1 \sqsubseteq_p \Gamma_2$$

which states that if a variable contained in p is not bound by the existing environment Γ_1, then it is bound to a fresh type variable in the new environment Γ_2. The relation is specified by the rules in Figure 4.3.

$$\frac{t_x : T \in \Gamma}{\Gamma \sqsubseteq_{t_x} \Gamma} \qquad \texttt{(ctx-id)}$$

$$\frac{t_x : T \notin \Gamma \qquad \Gamma' = (\Gamma,\ `a,\ t_x : t[`a])}{\Gamma \sqsubseteq_{t_x} \Gamma'} \qquad \texttt{(ctx-update)}$$

$$\frac{}{\Gamma \sqsubseteq_{t} \Gamma} \qquad \texttt{(ctx-leaf)}$$

$$\frac{\forall i.\ \Gamma_{i-1} \sqsubseteq_{p_i} \Gamma_i}{\Gamma_0 \sqsubseteq_{t[\![p_i^{i \in 1..n}]\!]} \Gamma_n} \qquad \texttt{(ctx-tree)}$$

Figure 4.3: Updating typing contexts with implicitly declared variables

Example To observe the effect of implicit declaration on an environment on concrete examples, consider the following match equations. Each equation is listed with the resulting typing context assuming the initially empty context $\oslash$. Note that these examples illustrate just the implicit declarations. Actual matching of variables and concrete terms will be discussed later.

1. $\mathrm{id}[\![\mathrm{a}]\!] = \mathrm{id}[\![\mathrm{a}]\!]$ does not change the context because both sides of the match equation are ground terms

2. id$_1$ = id⟦a⟧ updates the context to {id$_1$: id[‘x]}. Note that id$_1$ will become bound to id⟦a⟧ after the actual matching is completed.

3. expr⟦id$_x$+id$_y$⟧ = expr⟦a+b⟧ updates the context to {id$_x$: id[‘x], id$_y$: id[‘y]}

$$\frac{}{\Gamma \vdash \mathit{true} : \langle \mathit{bool}, \Gamma \rangle} \qquad \texttt{(m-true)}$$

$$\frac{}{\Gamma \vdash \mathit{false} : \langle \mathit{bool}, \Gamma \rangle} \qquad \texttt{(m-false)}$$

$$\frac{\Gamma \vdash m : \langle \mathit{bool}, \Gamma \rangle}{\Gamma \vdash \mathit{not}\ m : \langle \mathit{bool}, \Gamma \rangle} \qquad \texttt{(m-not)}$$

$$\frac{\Gamma \vdash m_1 : \langle \mathit{bool}, \Gamma_1 \rangle \qquad \Gamma_1 \vdash m_2 : \langle \mathit{bool}, \Gamma_2 \rangle}{\Gamma \vdash m_1\ \mathit{andalso}\ m_2 : \langle \mathit{bool}, \Gamma_2 \rangle} \qquad \texttt{(m-andalso)}$$

$$\frac{\Gamma \vdash m_1 : \langle \mathit{bool}, \Gamma_1 \rangle \qquad \Gamma \vdash m_2 : \langle \mathit{bool}, \Gamma_2 \rangle \qquad \Gamma_3 = \Gamma_1 \hat{\cap} \Gamma_2}{\Gamma \vdash m_1\ \mathit{orelse}\ m_2 : \langle \mathit{bool}, \Gamma_3 \rangle} \qquad \texttt{(m-orelse)}$$

$$\frac{\Gamma \sqsubseteq_p \Gamma_1 \qquad \Gamma_1 \vdash p : \langle T_1, \Gamma_2 \rangle \qquad \Gamma_2 \vdash s : \langle T_2, \Gamma_3 \rangle \qquad T_1 \Subset T_2}{\Gamma \vdash p = s : \langle \mathit{bool}, \Gamma_3 \rangle} \qquad \texttt{(m-match)}$$

Figure 4.4: Typing of matches

The rules for the calculation of types of pattern matches are summarized in Figure 4.4. Rules `(m-true)` and `(m-false)` assign type *bool* to the boolean constants. Rule `(m-not)` assigns a boolean type to a negation if the negation's operand has a boolean type. The environment produced by the analysis of negation needs to be the same as the one prior to the analysis. In other words, there cannot be match equations within a negation that create new local variables and both sides of any match equation must be ground. This is in correspondence with the evaluation semantics of the transformation language, where negation (including double negation) does not modify the existing bindings.[2]

Rule `(m-andalso)` assigns a boolean type to a conjunction of match expressions m_1 and m_2 if both of the expressions have a boolean type. The typing environment

[2] Prolog, for example, prohibits use of free variables within a negation. `single(X) :- person(X), not(married(X))` is valid, but `single(X) :- not(married(X))` is not.

is *threaded* through the analysis of conjunction such that the analysis of the second operand uses the environment produced by the analysis of the first.

Rule `(m-orelse)` specifies the analysis of a disjunction of match expressions. In contrast to the conjunction, the typing environment is not threaded through the analysis of the operands. Instead, analysis of both operands is initiated with the same environment Γ such that bindings of one operand do not interfere with the bindings of the other.

Example One of the use cases of a disjunction is to perform case analysis on terms, where different kinds of terms may lead to distinct bindings and execution paths. Consider as an example the following rewrite rule that analyzes the input term and rewrites it into the term's type:

```
expr$_1$  → id$_1$
  if {   ((expr$_1$ = expr⟦true⟧ orelse expr$_1$ = expr⟦false⟧)
          andalso id$_1$ = id⟦boolTy⟧)
       orelse
         (expr⟦integer$_1$⟧ = expr$_1$ andalso id$_1$ = id⟦intTy⟧)
  }
```

Here, if the input term is an expression that derives either `true` or `false`, then it is rewritten into a term `id` that derives `boolTy`. Otherwise, if the input term derives a term rooted in `integer`, then it is rewritten into id⟦intTy⟧. □

From the perspective of type analysis, since variables bound by the two operands of a disjunction may not be identical (for instance variable `integer`$_1$ in the example above), the only variables guaranteed to be bound after the evaluation of the disjunction's branches are variables bound by both branches. Otherwise, if the enclosing context refers to a variable bound by only one of the branches, then an error occurs at run-time if the execution passes through the branch that does not bind the variable. Therefore, the type system cannot allow an outer-context reference to a variable that is bound by only one of the branches of a disjunction. Such condition is flagged as a type error.

If a variable is bound by both operands of a disjunction, then the type system compares the two types. If the types are identical, then the resulting typing context binds the variable to that type. Otherwise, the variable in the resulting context is bound to the *least upper bound* or *join* of the two types, which is conceptually the largest common prefix of both types (see Section 4.7 for a discussion of joins).

In summary, the typing context resulting from the analysis of a disjunction is an intersection of the typing contexts produced by the analysis of its operands such that if the same variable is bound to distinct types in the two contexts, then the variable is bound to the join of the two types. We denote this extended intersection of contexts as $\hat{\cap}$. Thus, rule `(m-orelse)` calculates the resulting typing context as $\Gamma_3 = \Gamma_1 \hat{\cap} \Gamma_2$.

Finally, rule `(m-match)` assigns a boolean type to a match equation if, after updating the context with implicitly declared variables within the left-hand side operand ($\Gamma \sqsubseteq_p \Gamma_1$), the analysis of the left-hand side p produces type T_1 that is one of the possible types of s: i.e. $T_1 \Subset T_2$.

The need for membership check $\Subset$ arises due to the identity-based nature of the transformation language. Since the right-hand operand of a match equation can be an arbitrary strategy s (see Figure 3.1), the type of s can be a choice of possible types instead of just one type. Abstractly, if type of rule r is $T_1 \to T_2$ and type of term t is T_x, then the type of strategy $s = r\ t$ is T_2 if $T_x = T_1$ and T_x otherwise. To account for such multiple potential results of an application, we introduce union types $\cdot + \cdot$, which describe values drawn from heterogeneous types [60]. Thus, the type of strategy $s = r\ t$ can be calculated as $T_2 + T_x$.

$$\frac{T_1 = T_2}{T_1 \Subset T_2} \qquad \texttt{(m-eq-id)}$$

$$\frac{T_2 = T_{21} + T_{22} \qquad T_1 \Subset T_{21}}{T_1 \Subset T_2} \qquad \texttt{(m-eq-left)}$$

$$\frac{T_2 = T_{21} + T_{22} \qquad T_1 \Subset T_{22}}{T_1 \Subset T_2} \qquad \texttt{(m-eq-right)}$$

Figure 4.5: Membership of a type within a union type

The membership check $\Subset$ is specified by the rules summarized in Figure 4.5. Type T_1 is within another type T_2 if it is identical ($T_1 = T_2$) or if it is within one of the operands of the union type $T_2 = T_{21} + T_{22}$.

The need for introduction of union types deserves a special attention, because it plays a pivotal role in the kinds of errors that are detected by the type analysis for rewrite strategies. First, due to the static nature of the analysis, any static type system can only approximate the behavior of a program at run-time. This immediately implies that analysis of strategies is not *complete*, since some well-behaved programs may be rejected by the type system because the type system performs the analysis at the level of type abstractions and cannot check all possible execution scenarios for all possible inputs.

Second, in the classic type analysis view[60], any approximation must be *sound* to ensure that if a program is assigned a type, then it is guaranteed not to fail at run-time. If a program cannot be assigned a type, then it may or may not fail. Our type analysis takes a complement of this view in that if a program cannot be assigned a type, then it is guaranteed to fail at run-time. This perspective is in correspondence with the inherent nature of strategies, where a programmer, interested in rewriting specific kinds of terms, expects failure and provides additional rewrite rules to handle such failures. As an option of last resort, *ID* can be used as a catch-all handler of any failure. Therefore, in the strategic rewriting framework, the most immediate question is whether a strategy *can* succeed. Once this has been answered, the question of whether a strategy *will* succeed can be addressed by adding failure-handling rules as necessary, if at all (note that identity-based languages including *TL* implicitly handle failure by leaving the input term unchanged).

In summary, our type system defines an error as a strategy that *cannot* succeed, which is manifested as an empty set of possible types. In this perspective, the type analysis performs a conservative/sound approximation of whether a strategy can succeed.

Returning back to Figure 4.4, rule `(m-match)` states that a match equation has a type error if the type of the left-hand operand is not equal to any of the possible types of the right-hand operand.

Note that for readability, we abstract away from the details and denote type equality with operator $\cdot = \cdot$ or with identical type variables. In practice, type equality is enforced by

1. unification of the operands:
   e.g. function *unify:* $env \rightarrow ty \rightarrow ty \rightarrow env$;

2. application of the resulting substitution to the operands:
   e.g. function $applySubst : env \rightarrow ty \rightarrow ty$;

3. comparison of the resulting types for syntactic equality:
   e.g. function $eq : ty \rightarrow ty \rightarrow bool$.

We follow well-known implementations of unification and substitution (see [6] or p.327, § 22.4 in [60]) and hence avoid their repetition.

Example The following examples demonstrate concrete match expressions and the corresponding types assuming an empty initial context. Note that the recording of implicitly declared variables is omitted here, because it has been discussed previously.

- $id_1 = id\llbracket x \rrbracket$ is well-typed with context $\{id_1 : (id,\ [(idLex, [(x, [\ ]\ )]\ )]\ )\}$
- $expr_1 = id\llbracket x \rrbracket$ is a type error due to the mismatch of root terms
- $id\llbracket a \rrbracket = id\llbracket b \rrbracket$ is a type error due to mismatch of leaves
- $expr\llbracket id_1 \rrbracket = expr\llbracket x \rrbracket$ creates local term variable id_1 with context $\{id_1 : id[x]\}$
- $id_1 = id\llbracket x \rrbracket\ orelse\ id_1 = id\llbracket y \rrbracket$ is of type *bool* with context $\{id_1 : id[‘a]\}$

□

4.5 Typing of rewrite rules

To assign types to rewrite rules we use the type constructor $\cdot \rightarrow \cdot$, which captures the types of a rule's input term (left-hand side or premise) and output term or body. A rule's premise may contain variables, which are implicitly declared. Rules represent scoped abstractions in that any variable used by the output term must be declared and bound either by the rule's premise or by match equations within the rule's condition. Figure 4.6 summarizes the calculation of types for rewrite rules.

$$\frac{\Gamma \sqsubseteq_p \Gamma_1 \qquad \Gamma_1 \vdash p : \langle T_1, \Gamma_2 \rangle \qquad \Gamma_2 \vdash s : \langle T_2, \Gamma_3 \rangle}{\Gamma \vdash p \rightarrow s : \langle T_1 \rightarrow T_2, \Gamma_3 \rangle} \qquad \texttt{(rule-basic)}$$

$$\frac{\Gamma \sqsubseteq_p \Gamma_1 \qquad \Gamma_1 \vdash m : \langle bool, \Gamma_2 \rangle \qquad \Gamma_2 \vdash p : \langle T_1, \Gamma_3 \rangle \qquad \Gamma_3 \vdash s : \langle T_2, \Gamma_4 \rangle}{\Gamma \vdash p \rightarrow s\ if\,\{\ m\ \} : \langle T_1 \rightarrow T_2, \Gamma_4 \rangle} \qquad \texttt{(rule-cond)}$$

$$\frac{\Gamma' = (\Gamma, ‘x)}{\Gamma \vdash ID : \langle ‘x \rightarrow ‘x, \Gamma' \rangle} \qquad \texttt{(rule-id)}$$

$$\frac{\Gamma' = (\Gamma, ‘x)}{\Gamma \vdash SKIP : \langle ‘x \rightarrow ‘x, \Gamma' \rangle} \qquad \texttt{(rule-skip)}$$

Figure 4.6: Typing of rewrite rules

Rule `(rule-basic)` calculates the type of a rewrite rule that does not have a condition. Such rule is well-typed if both of its input and output terms are well-typed after an update of the initial context with variables contained within premise p. Rule (rule-cond) states that a conditional rewrite rule is well-typed if, in addition to the input and output terms, the rule's condition is also well-typed. The sequence of type analysis goes from the initial context update with variables in p to the analysis of match expressions in m and concludes with calculation of types for the input and output terms. This is to ensure that any type inferences made during the analysis of match expressions are propagated to the calculation of the type of the input term.

The built-in rewrite abstractions *ID* and *SKIP* are generic in that they are applicable to any input term and return the term unchanged. Therefore, rules (rule-id) and (rule-skip) pick a fresh type variable and assign type $`x \rightarrow `x$ to these abstractions.

Example The following are examples of rewrite rules and the types calculated based on the typing rules above:

- $\texttt{expr}_1 \rightarrow \texttt{expr}[\![\texttt{expr}_1\ \texttt{+ 1}]\!]$: $\texttt{expr[`a]} \rightarrow \texttt{expr[expr[`a] + 1]}$

- $\texttt{expr}_1 \rightarrow \texttt{expr}_2\ \texttt{if}\ \{\ \texttt{expr}[\![\texttt{expr}_2\ \texttt{+ 0}]\!]\ \texttt{=}\ \texttt{expr}_1\ \}$:

  $\texttt{expr[expr[`b] + 0]} \rightarrow \texttt{expr[`b]}$

- $\texttt{expr}_1 \rightarrow \texttt{id}_1$

  $\texttt{if}\ \{\quad \texttt{((expr}_1\ \texttt{=}\ \texttt{expr}[\![\texttt{true}]\!]\ \texttt{orelse}\ \texttt{expr}_1\ \texttt{=}\ \texttt{expr}[\![\texttt{false}]\!]\texttt{)}$

  $\qquad\qquad \texttt{andalso}\ \texttt{id}_1\ \texttt{=}\ \texttt{id}[\![\texttt{boolTy}]\!]\texttt{)}$

  $\qquad \texttt{orelse}$

  $\qquad\qquad \texttt{(expr}[\![\texttt{integer}_1]\!]\ \texttt{=}\ \texttt{expr}_1\ \texttt{andalso}\ \texttt{id}_1\ \texttt{=}\ \texttt{id}[\![\texttt{intTy}]\!]\texttt{)}$

  $\}$

  : $\texttt{expr['a]} \rightarrow \texttt{id[idLex['d]]}$

□

4.6 Typing of combinators

Combinators compose rewrite rules sequentially or conditionally. In sequential composition, the output of the first rule or strategy is used as input to the second. In conditional composition, the second rule or strategy is attempted on the input term only if the first strategy fails to apply to the input term; the output of the composition is the output of the first successful application.

4.6.1 Conditional composition

Figure 4.7 summarizes the typing rules for conditional composition. Rule `(comb-lcond)` calculates the type of left-conditional composition. The type of such composition is a union of the types of the operands provided that the right operand is reachable, which is checked by predicate `canReach`. A strategy in a conditional composition's sequence is reachable if it is not subsumed by other strategies earlier in the sequence. Figure 4.8 summarizes the specification of the predicate. In the base case, `(reach-rule)` states that rule x is reachable from another rule y, if the premise (input term) of rule x is not a subtype of the premise of rule y. The other rules – `(reach-sum-right)` and `(reach-sum-left)` – check reachability if either of the operands is a union type.

Analysis of reachable strategies makes use of the subtyping relation $\cdot <: \cdot$. Figure 4.9 summarizes the subtyping rules. Rule `(s-tyvar)` states that anything is a subtype of a type variable. Rule `(s-term)` states that a term is a subtype of another if the root symbols are identical and its sub-terms are subtypes of the other term's sub-terms. Rules `(ss-base)` and `(ss-tail)` check if a list of terms is a subtype of another list of terms, which is the case if each of the terms in the subtype list is a subtype of the corresponding term in the super-type list. Rule `(s-rule)` is equivalent to the subtyping rule for functions (p.184, [60]) in that subtyping is contravariant in arguments and covariant in outputs. Finally, rules `(s-refl)` and `(s-trans)` reflect that the subtyping relation is reflexive and transitive.

$$\frac{\Gamma \vdash s_1 : \langle T_1, \Gamma_1 \rangle \qquad \Gamma \vdash s_2 : \langle T_2, \Gamma_2 \rangle \qquad \mathit{canReach}(T_1, T_2)}{\Gamma \vdash s_1 \mathrel{\lhd\!\!+} s_2 : \langle T_1 + T_2, \Gamma \rangle} \qquad \texttt{(comb-lcond)}$$

$$\frac{\Gamma \vdash s_2 : \langle T_2, \Gamma_1 \rangle \qquad \Gamma \vdash s_1 : \langle T_1, \Gamma_2 \rangle \qquad \mathit{canReach}(T_2, T_1)}{\Gamma \vdash s_1 \mathrel{+\!\!\rhd} s_2 : \langle T_2 + T_1, \Gamma \rangle} \qquad \texttt{(comb-rcond)}$$

$$\frac{\Gamma \vdash s_1 : \langle T_1, \Gamma_1 \rangle \qquad \Gamma \vdash s_2 : \langle T_2, \Gamma_2 \rangle}{\Gamma \vdash s_1 \mathrel{\lhd\!\!+\!\!\rhd} s_2 : \langle T_1 + T_2, \Gamma \rangle} \qquad \texttt{(comb-acond)}$$

Figure 4.7: Typing rules for conditional combinators

$$\frac{\neg(c <: a)}{\mathit{canReach}(a \rightarrow b, c \rightarrow d)} \qquad \texttt{(reach-rule)}$$

$$\frac{(\mathit{canReach}(a, c) \wedge \mathit{canReach}(a, d))}{\mathit{canReach}(a, c + d)} \qquad \texttt{(reach-sum-right)}$$

$$\frac{(\mathit{canReach}(a, c) \wedge \mathit{canReach}(b, c))}{\mathit{canReach}(a + b, c)} \qquad \texttt{(reach-sum-left)}$$

Figure 4.8: Analysis of reachability. If y is reachable in composition $x \mathrel{\lhd\!\!+} y$, then $\mathit{canReach}(x, y)$ holds true.

Example To observe the analysis of reachable strategies, suppose we are given the following rules:

$$\texttt{a:expr}\llbracket\texttt{x}\rrbracket \;\rightarrow\; \texttt{expr}\llbracket\texttt{x-1}\rrbracket \qquad \texttt{b:expr}\llbracket\texttt{id}_1\rrbracket \;\rightarrow\; \texttt{expr}\llbracket\texttt{id}_1\texttt{+1}\rrbracket$$

- Strategy (*`ID`* $\mathrel{\lhd\!\!+}$ `a`) is a type error, because *`ID`* successfully rewrites any term, which makes rule `a` unreachable. This is checked at the level of types by determining that the input type of rule `a` (`expr[x]`) is a subtype of the input type of *`ID`* (‘a), which does not fit the premise of the typing rule `(reach-rule)`.

$$\frac{}{S <: \text{`}x} \qquad \texttt{(s-tyvar)}$$

$$\frac{S = T \qquad Ss\ [<:]\ Ts}{(S, Ss) <: (T, Ts)} \qquad \texttt{(s-term)}$$

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \qquad \texttt{(s-rule)}$$

$$\frac{}{S <: S} \qquad \texttt{(s-refl)}$$

$$\frac{S <: U \qquad U <: T}{S <: T} \qquad \texttt{(s-trans)}$$

$$\frac{}{[\,]\ [<:]\ [\,]} \qquad \texttt{(ss-base)}$$

$$\frac{S <: T \qquad Ss[<:]Ts}{S :: Ss\ [<:]\ T :: Ts} \qquad \texttt{(ss-tail)}$$

Figure 4.9: Analysis of subtyping.

- Strategy (b $\lhd\!\!+$ a) is a type error, because input term expr$\llbracket$x$\rrbracket$, which is the only possible term that satisfies the input pattern of rule a, is also matched by the input pattern of rule b, which makes rule a unreachable. At the level of types, this is determined by checking that sub-term id$\llbracket$x$\rrbracket$ of term expr$\llbracket$x$\rrbracket$ is a subtype of sub-term $\texttt{id}_1$ of term expr$\llbracket\texttt{id}_1\rrbracket$.

- Strategy (a $\lhd\!\!+$ b) is well-typed with type

  expr[x] $\to$ expr[x - 1] + expr[id[‘a]] $\to$ expr[id[‘a] + 1].

□

Continuing with the rules in Figure 4.7, rule (comb-rcond) performs the same analysis as in the case of left-conditional composition except that the order of analysis and reachability is reversed. Finally, rule (comb-acond) states the the type of non-deterministic conditional composition is the union of the types of its operands. Since the choice of which operand is executed first is non-deterministic, the reachability analysis is skipped.

4.6.2 Strict sequence

Figure 4.10 summarizes the typing rules for strict sequential composition. Both left- and right-biased compositions are analyzed by first computing the types of the operands and invoking further analysis of the types using function `ckStar`, which given operands' types and an environment returns the set of possible types and the environment of the entire composition.

$$\frac{\Gamma \vdash s_a : \langle T_a, \Gamma_a \rangle \qquad \Gamma_a \vdash s_b : \langle T_b, \Gamma_b \rangle \\ \Gamma_b \vdash ckStar(T_a, T_b) : \langle \{T_i^{i \in 1..n : n \geq 1}\}, \Gamma_c \rangle}{\Gamma \vdash s_a {<\!\!*}\, s_b : \langle T_1 + T_2 + \ldots + T_n, \Gamma_c \rangle} \qquad \texttt{(comb-lstar)}$$

$$\frac{\Gamma \vdash s_b : \langle T_b, \Gamma_b \rangle \qquad \Gamma_b \vdash s_a : \langle T_a, \Gamma_a \rangle \\ \Gamma_a \vdash ckStar(T_b, T_a) : \langle \{T_i^{i \in 1..n : n \geq 1}\}, \Gamma_c \rangle}{\Gamma \vdash s_a {*\!\!>} s_2 : \langle T_1 + T_2 + \ldots + T_n, \Gamma_c \rangle} \qquad \texttt{(comb-rstar)}$$

Figure 4.10: Typing rules for strict sequence

Figure 4.11 specifies the function `ckStar`. In the base case, which is checked by `(ckstar-rule`$^+$`)`, strict sequential composition is well-typed if the output type of the first rule in the sequence is the same as (or type-unifiable with) the input type of the second rule. This is in accordance with the evaluation semantics of strict sequence, which succeeds only if the first strategy successfully applies to the input term and the second strategy successfully applies to the output term of the first strategy. If the types are not the same, which is indicated by `(ckstar-rule`$^-$`)`, then the composition has no chance of succeeding, because the second strategy will always fail to apply. In this case, the set of possible output types is empty.

The other cases check the compatibility of types when one or both of the operands has a union type. For example, if the second operand has a union type, then rule `(ckstar-sum-right)` recursively invokes the analysis on composition of the first operand ($T_a \rightarrow T_b$) with each of the union type's operands T_c and T_d. The resulting sets of possible types S_{abc} and S_{abd} are combined using the standard union operation:

$$\frac{T_b = T_c}{\Gamma \vdash \mathit{ckStar}(T_a \rightarrow T_b, T_c \rightarrow T_d) : \langle \{T_a \rightarrow T_d\}, \Gamma \rangle} \qquad \texttt{(ckstar-rule}^{+}\texttt{)}$$

$$\frac{T_b \neq T_c}{\Gamma \vdash \mathit{ckStar}(T_a \rightarrow T_b, T_c \rightarrow T_d) : \langle\ \emptyset, \Gamma \rangle} \qquad \texttt{(ckstar-rule}^{-}\texttt{)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{ckStar}(T_a \rightarrow T_b, T_c) : \langle S_{abc}, \Gamma_1 \rangle \\ \Gamma \vdash \mathit{ckStar}(T_a \rightarrow T_b, T_d) : \langle S_{abd}, \Gamma_2 \rangle\end{array}}{\Gamma \vdash \mathit{ckStar}(T_a \rightarrow T_b, T_c + T_d) : \langle S_{abc} \cup S_{abd}, \Gamma \rangle} \qquad \texttt{(ckstar-sum-right)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{ckStar}(T_a, T_c \rightarrow T_d) : \langle S_{acd}, \Gamma_1 \rangle \\ \Gamma \vdash \mathit{ckStar}(T_b, T_c \rightarrow T_d) : \langle S_{bcd}, \Gamma_2 \rangle\end{array}}{\Gamma \vdash \mathit{ckStar}(T_a + T_b, T_c \rightarrow T_d) : \langle S_{acd} \cup S_{bcd}, \Gamma \rangle} \qquad \texttt{(ckstar-sum-left)}$$

$$\frac{\begin{array}{cc}\Gamma \vdash \mathit{ckStar}(T_a, T_c) : \langle S_{ac}, \Gamma_1 \rangle & \Gamma \vdash \mathit{ckStar}(T_a, T_d) : \langle S_{ad}, \Gamma_2 \rangle \\ \Gamma \vdash \mathit{ckStar}(T_b, T_c) : \langle S_{bc}, \Gamma_3 \rangle & \Gamma \vdash \mathit{ckStar}(T_b, T_d) : \langle S_{bd}, \Gamma_4 \rangle\end{array}}{\Gamma \vdash \mathit{ckStar}(T_a + T_b, T_c + T_d) : \langle S_{ac} \cup S_{ad} \cup S_{bc} \cup S_{bd}, \Gamma \rangle} \qquad \texttt{(ckstar-sum-both)}$$

Figure 4.11: Analysis of operands of strict sequence

$S_{abc} \cup S_{abd}$. If one of the operands is a type error, then the resulting set of types is that of the other operand. If both operands are type errors, then an empty set is returned. If the initial invoker of *ckStar* receives an empty set of possible types, then a type error occurs because the typing rules `(comb-lstar)` and `(comb-rstar)` expect a non-empty set of possible types.

Example The following examples list strategies and their corresponding types computed according to the rules discussed above:

- `expr⟦expr`$_1$` + 0⟧` → `expr`$_1$ <* `expr⟦x⟧` → `expr⟦x+1⟧ :`

  `expr[x + 0]` → `expr[x + 1]`

- `expr⟦id`$_1$` + 0⟧` → `id`$_1$ <* `expr⟦x⟧` → `expr⟦x+1⟧ :  type error`

- `id⟦a⟧` → `id⟦b⟧` <* `(id⟦a⟧` → `id⟦d⟧` <+ `id⟦b⟧` → `id⟦c⟧) :`

  `id[a]` → `id[c]`

- `id⟦a⟧` → `id⟦c⟧` <+ `id⟦b⟧` → `id⟦c⟧)` <*

  `(id⟦a⟧` → `id⟦d⟧` <+ `id⟦b⟧` → `id⟦d⟧) :  type error`

4.6.3 Non-strict sequence

The non-strict sequential combinators $<;$ and $;>$ compose two strategies such that the second strategy is applied on the output term of the first strategy irrespective of whether or not the first strategy succeeded on the input term. This allows a programmer to incrementally collect several strategies and apply them sequentially without having to check that all of them can succeed. Figure 4.12 summarizes the typing rules for these combinators. Similar to the typing rules of Figure 4.10, the analysis proceeds by first computing the types of operand strategies and then passing the types to function *ckSeq*, which performs further type analysis.

$$\frac{\Gamma \vdash s_a : \langle T_a, \Gamma_a \rangle \qquad \Gamma_a \vdash s_b : \langle T_b, \Gamma_b \rangle \qquad \Gamma_b \vdash \mathit{ckSeq}(T_a, T_b) : \langle \{T_i^{i \in 1..n:n \geq 1}\}, \Gamma_c \rangle}{\Gamma \vdash s_a {<;} s_b : \langle T_1 + T_2 + \ldots + T_n, \Gamma_c \rangle} \quad \texttt{(comb-lseq)}$$

$$\frac{\Gamma \vdash s_b : \langle T_b, \Gamma_b \rangle \qquad \Gamma_b \vdash s_a : \langle T_a, \Gamma_a \rangle \qquad \Gamma_a \vdash \mathit{ckSeq}(T_b, T_a) : \langle \{T_i^{i \in 1..n:n \geq 1}\}, \Gamma_c \rangle}{\Gamma \vdash s_a {;>} s_b : \langle T_1 + T_2 + \ldots + T_n, \Gamma_c \rangle} \quad \texttt{(comb-rseq)}$$

Figure 4.12: Typing rules for non-strict sequence

Figure 4.13 summarizes the definition of *ckSeq*. According to `(ckseq-rule`$^+$`)`, if the output type of the first operand (T_b) is the same as (or type-unifiable with) the input type of the second operand (T_c), then there are three possible execution paths through non-strict sequential composition:

1. Both of the strategies succeed, in which case the type of the composition is $T_a \to T_d$;

2. The first strategy fails to apply on the input term, in which case the second strategy will be attempted on the input term. The type of the second strategy becomes one of the possible types of the composition: $T_c \to T_d$.

3. Due to unification and propagation of type bindings, the type of the first

strategy may become more constrained. For example, composition $\mathtt{expr}[\![\mathtt{id}_1]\!] \rightarrow \mathtt{id}_1 <; \mathtt{id}[\![\mathtt{x}]\!] \rightarrow \mathtt{id}[\![\mathtt{y}]\!]$, using the two executions paths above, has type `expr[x]` $\rightarrow$ `id[y] + id[x]` $\rightarrow$ `id[y]`. Since the first strategy is applicable to terms other than $\mathtt{expr}[\![\mathtt{x}]\!]$, it is possible that only the first strategy will succeed. Therefore, the type of the composition has the type of the first strategy as one of its possible types. In the example, it is `expr[id['a]]` $\rightarrow$ `id['a]`.

Note that the implementation of the type system performs additional checks to remove duplicate types from union types and to reduce the number of possible types whenever possible: e.g. when the second strategy is guaranteed to apply to the output of the first strategy.

$$\frac{T_b = T_c}{\Gamma \vdash \mathit{ckSeq}(T_a \rightarrow T_b, T_c \rightarrow T_d) : \langle \{T_a \rightarrow T_d, T_a \rightarrow T_b, T_c \rightarrow T_d\}, \Gamma \rangle} \quad \texttt{(ckseq-rule}^{+}\texttt{)}$$

$$\frac{T_b \neq T_c \qquad \mathit{canReach}(T_a \rightarrow T_b, T_c \rightarrow T_d)}{\Gamma \vdash \mathit{ckSeq}(T_a \rightarrow T_b, T_c \rightarrow T_d) : \langle \{T_a \rightarrow T_b, T_c \rightarrow T_d\}, \Gamma \rangle} \quad \texttt{(ckseq-rule}^{-}\texttt{)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{ckSeq}(T_a \rightarrow T_b, T_c) : \langle S_{abc}, \Gamma_1 \rangle \\ \Gamma \vdash \mathit{ckSeq}(T_a \rightarrow T_b, T_d) : \langle S_{abd}, \Gamma_2 \rangle\end{array}}{\Gamma \vdash \mathit{ckSeq}(T_a \rightarrow T_b, T_c + T_d) : \langle S_{abc} \cup S_{abd}, \Gamma \rangle} \quad \texttt{(ckseq-sum-right)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{ckSeq}(T_a, T_c \rightarrow T_d) : \langle S_{acd}, \Gamma_1 \rangle \\ \Gamma \vdash \mathit{ckSeq}(T_b, T_c \rightarrow T_d) : \langle S_{bcd}, \Gamma_2 \rangle\end{array}}{\Gamma \vdash \mathit{ckSeq}(T_a + T_b, T_c \rightarrow T_d) : \langle S_{acd} \cup S_{bcd}, \Gamma \rangle} \quad \texttt{(ckseq-sum-left)}$$

$$\frac{\begin{array}{cc}\Gamma \vdash \mathit{ckSeq}(T_a, T_c) : \langle S_{ac}, \Gamma_1 \rangle & \Gamma \vdash \mathit{ckSeq}(T_a, T_d) : \langle S_{ad}, \Gamma_2 \rangle \\ \Gamma \vdash \mathit{ckSeq}(T_b, T_c) : \langle S_{bc}, \Gamma_3 \rangle & \Gamma \vdash \mathit{ckSeq}(T_b, T_d) : \langle S_{bd}, \Gamma_4 \rangle\end{array}}{\Gamma \vdash \mathit{ckSeq}(T_a + T_b, T_c + T_d) : \langle S_{ac} \cup S_{ad} \cup S_{bc} \cup S_{bd}, \Gamma \rangle} \quad \texttt{(ckseq-sum-both)}$$

Figure 4.13: Analysis of operands of non-strict sequence

If the input and output types are not compatible, then there are only two possible paths through the composition: either the first or the second strategy. In addition to assigning the union of the two types as the output type of the composition, rule (`ckseq-rule`$^{-}$) states that the second strategy must be reachable, otherwise the second strategy has no chance of succeeding, which is obviously a programming error.

Besides the detection of incompatible input and output types, the type system also issues an informational log notice to the standard console notifying a programmer that the non-strict sequential combinator can be replaced by one of the conditional combinators. In other words, this condition manifests equivalence of $s_1 {<\!\!;} s_2$ to $s_1 {<\!\!\!+} s_2$ and $s_1 {;\!\!>} s_2$ to $s_1 {+\!\!\!>} s_2$.

The remaining rules `(ckseq-sum-right)`, `(ckseq-sum-left)` and `(ckseq-sum-both)` analyze the composition when one or both operands is a union type. Each of a union type's operands is composed with the type of the other strategy and results are combined into a union of possible output types.

Example Here are example of strategies and their types:

- $\mathtt{id}\llbracket \mathtt{x} \rrbracket \ \rightarrow\ \mathtt{id}\llbracket \mathtt{y} \rrbracket \ {<\!\!;}\ \mathtt{id}\llbracket \mathtt{y} \rrbracket \ \rightarrow\ \mathtt{id}\llbracket \mathtt{z} \rrbracket$:

  $\mathtt{id[x]} \ \rightarrow\ \mathtt{id[z]}\ \mathtt{+}\ \mathtt{id[y]} \ \rightarrow\ \mathtt{id[z]}$

- $\mathtt{id}\llbracket \mathtt{y} \rrbracket \ \rightarrow\ \mathtt{id}\llbracket \mathtt{z} \rrbracket \ {<\!\!;}\ \mathtt{id}\llbracket \mathtt{x} \rrbracket \ \rightarrow\ \mathtt{id}\llbracket \mathtt{y} \rrbracket$:

  $\mathtt{id[y]} \ \rightarrow\ \mathtt{id[z]}\ \mathtt{+}\ \mathtt{id[x]} \ \rightarrow\ \mathtt{id[y]}$

- $\mathtt{id}_1 \ \rightarrow\ \mathtt{id}\llbracket \mathtt{z} \rrbracket \ {<\!\!;}\ \mathtt{id}\llbracket \mathtt{x} \rrbracket \ \rightarrow\ \mathtt{id}\llbracket \mathtt{y} \rrbracket$: `type error`

4.7 Analysis of application

Having assigned types to the core constructs of rewrite strategies – patterns, match expressions, rewrite rules and combinators, we are now ready to discuss the analysis of applying a strategy to a term. This rounds out the discussion of the core features of the type system. Other features of rewrite strategies such as iterators and non-standard features of the target language such as higher-order rules and compositions will be considered as extensions of this core system.

Application of strategy s to term t denoted by $s\ t$ is well-typed if type of t is a term type and type of s contains an arrow type either directly or as part of a union type.

Figure 4.14 summarizes the typing rules of analysis of application. In particular, the types of constituents of an application are computed and passed onto function *reduce*, which performs further analysis at the level of types.

$$\frac{\Gamma \vdash s : \langle T_a, \Gamma_a \rangle \qquad \Gamma_a \vdash t : \langle T_b, \Gamma_b \rangle \qquad \Gamma_b \vdash \mathit{reduce}(T_a, T_b) : \langle \{T_i^{i\in 1..n:n\geq 1}\}, \Gamma_c \rangle}{\Gamma \vdash s\ t : \langle T_1 + T_2 + \ldots + T_n, \Gamma_c \rangle} \qquad \texttt{(app)}$$

$$\frac{}{\Gamma \vdash \mathit{reduce}(T_a \rightarrow T_b, T_a) : \langle \{T_b, T_a\}, \Gamma \rangle} \qquad \texttt{(red-rule)}$$

$$\frac{\Gamma \vdash \mathit{reduce}(T_a, T_c) : \langle S_{ac}, \Gamma_1 \rangle \qquad \Gamma \vdash \mathit{reduce}(T_b, T_c) : \langle S_{bc}, \Gamma_2 \rangle \qquad \Gamma_3 = \Gamma_1 \hat{\cap} \Gamma_2}{\Gamma \vdash \mathit{reduce}(T_a + T_b, T_c) : \langle S_{ac} \cup S_{bc}, \Gamma_3 \rangle} \qquad \texttt{(red-sum)}$$

$$\frac{\Gamma, ‘a, ‘b, ‘x : ‘a \rightarrow ‘b \vdash \mathit{reduce}(‘a \rightarrow ‘b, T_c) : \langle S_{abc}, \Gamma_1 \rangle}{\Gamma \vdash \mathit{reduce}(‘x, T_c) : \langle S_{abc}, \Gamma_1 \rangle} \qquad \texttt{(red-tyvar)}$$

Figure 4.14: Analysis of strategy application

Rule `(red-rule)` states that application of arrow type $T_a \rightarrow T_b$ to a type is well-formed if the type is the same as input type T_a of the arrow type. Further, since application may fail due to non-type-related issues (e.g. the body of a conditional rewrite rule evaluates to false) and the application failure has identity-based behavior, which leaves the input term unchanged, the set of possible output types includes the input term's type. Therefore, the set of possible types of applying an arrow type is $\{T_b, T_a\}$.

Rule `(red-tyvar)` performs type-inferencing by expanding the type variable into arrow type $‘a \rightarrow ‘b$, where variables $‘a$ and $‘b$ are fresh, and binding variable $‘x$ to the arrow type prior to recursive invocation $\mathit{reduce}(‘a \rightarrow ‘b, T_c)$.

Finally, rule `(red-sum)` checks application of a union type by recursively invoking *reduce* on the union type's operands and combining the resulting sets of types S_{ac}, S_{bc} into a union. Since the two recursive invocations may bind the same variable to distinct types, the resulting typing contexts Γ_1 and Γ_2 are joined by performing an extended intersection operation $\hat{\cap}$, such that if variable x is bound to T_{x1} in Γ_1 and to T_{x2} in Γ_2 with $T_{x1} \neq T_{x2}$, then it is bound to the join type $T_{x1} \vee T_{x2}$ in $\Gamma_1 \hat{\cap} \Gamma_2$. In all other cases, the behavior of $\hat{\cap}$ is that of the standard set intersection operation.

Figure 4.15 specifies the calculation of join types. Calculation of joins for arrow types requires calculation of meets [60]. Therefore, the two algorithms are listed simultaneously as mutually recursive definitions. We write $S \vee T = J$ for "J is the join of S and T" and $S \wedge T = M$ for "M is the meet of S and T".

In particular, if one of the operands is a subtype of the other, then the join is the super-type. If neither type is a subtype of the other, we search for some type $S' = \ \Uparrow S$ that is more generic than the given type S. If the calculated type S' is indeed more generic, then we recursively invoke the calculation of join $T \vee S' = J$ and use J as the result. Otherwise, if $S' = S$, that is S is already in the most generic form, then we find a more generic type $T' = \ \Uparrow T$ and use T' in recursive calculation of J.

If the given types are arrow types, then, to be consistent with the subtype relation for arrow types, we compute the meet for input types – $M = S_1 \wedge T_1$ – and join for output types – $J = S_2 \vee T_2$. The resulting join type is $M \to J$. Finally, it is possible that the calculation of meets may fail, because the existence of meets is not guaranteed by the subtype relation. In this case, the join of such types is the (fresh) type variable $`a$ – the most generic type.

Calculation of meets mirrors the calculation of joins. In particular, if one of the operands is a subtype of the other, then the meet is the subtype. If the operands are arrow types, then the meet is $J \to M$, where J is the join of input types and M is the meet of output types. If none of the above holds, then there does not exist a meet and the calculation fails.

Figure 4.16 summarizes the calculation of a more generic type $\Uparrow T_i$ given some type T_i. In the base case, the more generic type of a type variable is the type variable itself – rule `(up-tyvar)`. To compute a more generic type of a term type (T, Ts),

$$S \vee T = \left\{\begin{array}{ll} T & \text{if} \quad S <: T \\ S & \text{if} \quad T <: S \\ J & \text{if} \quad \Uparrow S = S' \\ & \phantom{\text{if}} \quad S' \neq S \\ & \phantom{\text{if}} \quad T \vee S' = J \\ J & \text{if} \quad \Uparrow S = S' \\ & \phantom{\text{if}} \quad S' = S \\ & \phantom{\text{if}} \quad \Uparrow T = T' \\ & \phantom{\text{if}} \quad T' \vee S' = J \\ M \to J & \text{if} \quad S = S_1 \to S_2 \\ & \phantom{\text{if}} \quad T = T_1 \to T_2 \\ & \phantom{\text{if}} \quad S_1 \wedge T_1 = M \\ & \phantom{\text{if}} \quad S_2 \vee T_2 = J \\ `a & \text{otherwise} \end{array}\right.$$

$$S \wedge T = \left\{\begin{array}{ll} S & \text{if} \quad S <: T \\ T & \text{if} \quad T <: S \\ J \to M & \text{if} \quad S = S_1 \to S_2 \\ & \phantom{\text{if}} \quad T = T_1 \to T_2 \\ & \phantom{\text{if}} \quad S_1 \vee T_1 = J \\ & \phantom{\text{if}} \quad S_2 \wedge T_2 = M \\ \mathit{fail} & \text{otherwise} \end{array}\right.$$

Figure 4.15: Calculation of join (and meet) types

we first make the list of immediate children Ts more generic. If the resulting list is not the same as the original – rule `(up-term-children)`, then the more generic type is (T, Ts'). Otherwise – rule `(up-term-root)`, the more generic type is a fresh type variable $`a$.

To compute a generic type of a list of types, we make the head of a list, if any, more generic – rule `(up-list-head)`, and otherwise make the tail of the list more generic – rule `(up-list-tail)`.

Example The following examples demonstrate the analysis of application

- $(\texttt{expr}[\![\texttt{expr}_1\ \texttt{+}\ \texttt{0}]\!] \ \rightarrow \texttt{expr}_1)\ \texttt{expr}[\![\texttt{x}\ \texttt{+}\ \texttt{0}]\!]$:

  `expr[ x ] + expr[ x + 0 ]`

- $(\texttt{expr}[\![\texttt{expr}_1\ \texttt{+}\ \texttt{0}]\!] \ \rightarrow \texttt{expr}_1)\ \texttt{expr}[\![\texttt{x}\ \texttt{+}\ \texttt{1}]\!]$: `type error`

- $(\texttt{expr}[\![\texttt{expr}_1\ \texttt{+}\ \texttt{0}]\!] \ \rightarrow \texttt{expr}_1 \ \mathrel{\lhd\!\!\!+}\ \texttt{expr}[\![\texttt{expr}_1\ \texttt{*}\ \texttt{1}]\!] \ \rightarrow \texttt{expr}_1)$:

$$\frac{}{\Uparrow {‘}a = {‘}a} \qquad \texttt{(up-tyvar)}$$

$$\frac{[\Uparrow]Ts = Ts' \qquad Ts \neq Ts'}{\Uparrow (T, Ts) = (T, Ts')} \qquad \texttt{(up-term-children)}$$

$$\frac{[\Uparrow]Ts = Ts' \qquad Ts = Ts'}{\Uparrow (T, Ts) = {‘}a} \qquad \texttt{(up-term-root)}$$

$$\frac{}{[\Uparrow]\ [\,] = [\,]} \qquad \texttt{(up-list-base)}$$

$$\frac{\Uparrow T = T' \qquad T \neq T'}{[\Uparrow]\ [T :: Ts] = [T' :: Ts]} \qquad \texttt{(up-list-head)}$$

$$\frac{\Uparrow T = T' \qquad T = T' \qquad [\Uparrow]Ts = Ts'}{[\Uparrow]\ [T :: Ts] = [T :: Ts']} \qquad \texttt{(up-list-tail)}$$

Figure 4.16: Extending a type into a more generic type

`expr[ expr[ ’a ] + 0 ]` → `expr[ ’a ] +`

`expr[ expr[ ’b ] * 1 ]` → `expr[ ’b ]`

- (expr$\llbracket$expr$_1$ + 0$\rrbracket$ → expr$_1$ $\lhd\!\!+$

  expr$\llbracket$expr$_1$ * 1$\rrbracket$ → expr$_1$) expr$\llbracket$x + 0$\rrbracket$:

  `expr[ x ] + expr[x + 0]`

- expr$_x$ →

  (expr$\llbracket$expr$_1$ + 0$\rrbracket$ → expr$_1$ $\lhd\!\!+$ expr$\llbracket$expr$_1$ * 1$\rrbracket$ → expr$_1$) expr$_x$:

  `expr[ ’f ’g intLex[ ’h ] ]` → `(expr[ ’a ] + expr[ ’b ])`

4.8 Analysis of strategy declarations

A program in *TL* is a list of labeled strategy definitions (or rewrite abstractions) of the form $x : s$. The typing rules for programs are listed in Figure 4.17. A strategy definition is well-typed, if the definition's body is well-typed. If a definition is well-typed, then the original typing context is extended with the type binding for the new label (rule `(def-abs)`). A program is well-typed if all constituent rewrite abstractions are well-typed.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : \langle T, \Gamma \rangle} \qquad \texttt{(def-var)}$$

$$\frac{\Gamma \vdash s : \langle T, \Gamma_1 \rangle \quad \Gamma_2 = \Gamma_1, x : T}{\Gamma \vdash x : s : \langle T, \Gamma_2 \rangle} \qquad \texttt{(def-abs)}$$

$$\frac{\Gamma \vdash d_1 : \langle T_1, \Gamma_1 \rangle \quad \Gamma_1 \vdash d_2 : \langle T_2, \Gamma_2 \rangle}{\Gamma \vdash d_1\ d_2 : \langle T_2, \Gamma_2 \rangle} \qquad \texttt{(def-seq)}$$

Figure 4.17: Analysis of strategies and programs

4.9 Typing Properties

In the previous sections, we have presented the core parts of the type analysis: i.e. types of terms and patterns, analysis of rewrite rules, rule compositions and their applications. In this section, we summarize the properties of the core of type analysis, in particular the soundness of the analysis.

A type system is sound if well-typed terms do not go wrong. In the strategic rewriting setting, going wrong means applying a top-level strategy to a term, for which it is undefined. In *TL* this means applying the `main` strategy to an input term and always failing: i.e. $\texttt{main t} = \langle$ t, false$\rangle$ for all `t`. Note that constituent (internal) strategy failures may be handled by an enclosing composite strategy and thus applications of the form $(\texttt{t}_1 \rightarrow \texttt{t}_1\ \{\ \texttt{if false}\}\ \lessdot\!\!+\ \textit{ID}\ )\ \texttt{t}$ are well-typed even though there exist intermediate application failures: i.e. $(\texttt{t}_1 \rightarrow \texttt{t}_1\ \{\ \texttt{if false}\})$ `t`. Thus, a strategy goes wrong when the failure always bubbles up and escapes the strategic controls un-handled.

We formulate the following properties and leave their proof for all (≈70) constructors of the *TL* language for future work. A type soundness proof for a smaller language *StratCore* (5 constructors) along with its small-step operational semantics has been previously presented and discussed in [50]. Additional details can also be found in [49].

Lemma (Rewrite soundness): Given rewrite rule r of the form $lhs \rightarrow rhs$ and term t, if type of application $r\ t$ is $\{T_{rhs'}\}$, then evaluation of $r\ t$ succeeds and produces rhs', where $rhs' : T_{rhs'}$. If the type of application $r\ t$ is $\emptyset$, then the evaluation of the rewrite rule application cannot succeed.

This lemma states that the type analysis follows the standard rewriting semantics [6] such that if a substitution σ can be constructed from the match of `lhs` and `t` to build the output term σ`(rhs) = rhs'`, then the type system will correctly compute the type of the application result to be the set containing the type of `rhs'`. Further, if a substitution cannot be constructed, then the type system will identify this application failure by an empty set.

Theorem (Type soundness): If $s\ t : \{T_i^{i \in 1..n, n \geq 1}\}$, then evaluation of $s\ t$ succeeds and produces t' such that $t' : T'$ and $T' \in \{T_i^{i \in 1..n, n \geq 1}\}$.

The theorem lifts the type soundness of a single rule analysis to the type soundness of a strategy analysis. This theorem can be proved by a case-based analysis of the constructors of strategy s [50].

Chapter 5

Extension: Analysis of Other Strategic Features

In the previous chapter, we discussed the core of the type system that focused on the analysis of rewrite rules and their compositions. In this chapter, we continue the analysis of the transformation language *TL* by considering other features such as calls to Standard ML (SML) functions, higher-order rules and their compositions.

5.1 SML functions

In *TL* , a conditional rewrite rule may invoke an SML function to perform some computation that may not be easily achievable using rewrite strategies: e.g. imperative side effects such as modification of memory state and output to standard console. While the typical context of function invocation is a boolean-valued match expression, the results of a function call may also be matched to a pattern in a match equation:

$$p = sml.id(p^*)$$

where the (possibly empty) list of function arguments can contain arbitrary patterns. In other words, there is no restriction on the input and output terms of invocable functions. Therefore, to ensure the consistency of type analysis in the presence of function calls, it has been decided to allow a programmer to leverage type-checking and include function calls in the type analysis. For this purpose, a programmer can declare the type signature of a function in a signature block of a *TL* program. Having declared the signature of a function, the programmer can rely on the type system to ensure that the function is invoked with parameters of proper type and that the result of the function is used in a context of proper type.

In order to retain backward compatibility with existing *TL* programs and to provide programming flexibility, the declaration of function signatures is optional. If a call is made to a function, whose signature has not been declared, then the type system performs a "best-effort analysis" by type inferencing both the function arguments and the return type and ensuring that multiple invocations of the same function use arguments and results of the same respective types.

d	::=	...	*declarations:*
		"UserDefinedFunctions" "=" "sig" [sigs] "end"	*signature block*
sigs	::=	sig [sigs]	*signature*
sig	::=	id ":" typeExpr	*type declaration*
typeExpr	::=	typeTerm "→" typeBase	*arrow type*
typeTerm	::=	typeBase "*" typeTerm	*product type*
		typeBase	*base type*
typeBase	::=	"<"id">"	*schema type*
		"unit"	*unit type*
		"int"	*integer type*
		"bool"	*boolean type*
		"string"	*string type*
		"real"	*real type*

Figure 5.1: SML function type declarations

Figure 5.1 summarizes the grammar of signature declarations. The signature block consists of a list of signatures. Signature of a function is an arrow type, where the input type can be (a product of) one of the basic types or a schema type. A valid schema type has an identifier of one of non-terminal symbols of the term language's grammar.

To analyze the calls to declared functions, the type system performs a pre-processing step where the initially empty typing environment is populated with the type signatures of declared SML functions. The updated context is then used in the analysis of the program and the function calls.

Figure 5.2 summarizes the typing rules used in the analysis of function calls. We distinguish two kinds of function calls: calls with an empty and non-empty argument lists. In the first case, which is expressed by typing rules `(sml-call0-decl)` and `(sml-call0-infer)`, the type signature of the invoked function is looked up in the typing context. If the signature is found, then the resulting type is the function's return type. Otherwise, an inferencing step is made by picking a fresh type variable and binding it to the function's return type.

If a call is made to a function with parameters, then the signature of the function is looked up in the typing context. If one is found – rule `(sml-call-decl)`, then each of the function's actual arguments p_i is analyzed for its type. Further, for each i, the type of formal parameter i needs to match the type of actual parameter p_i. If so, the returned type is the function's declared return type.

On the other hand, if the invoked function's signature is not found in the typing context – rule `(sml-call-infer)`, then an inferencing step is made by binding the function's identifier to an arrow type with the matching number of formal parameters such that each component of the arrow type is a fresh type variable. Then, each of the actual parameters p_i is analyzed for its type T_i. Finally, each of the type variables introduced in the inferencing step is bound to the corresponding actual type T_i and the type $`m$ is returned by the analysis of the function call.

$$\frac{id : Unit \rightarrow T \in \Gamma_0}{\Gamma_0 \vdash sml.id() : \langle T, \Gamma_1 \rangle} \qquad \texttt{(sml-call0-decl)}$$

$$\frac{id : Unit \rightarrow T \notin \Gamma_0 \qquad \Gamma_1 = \Gamma_0, `a, id : Unit \rightarrow `a}{\Gamma_0 \vdash sml.id() : \langle `a, \Gamma_1 \rangle} \qquad \texttt{(sml-call0-infer)}$$

$$\frac{id : (T_i^{i \in 1..n, n \geq 1}) \rightarrow T \in \Gamma_0 \qquad \Gamma_{i-1} \vdash p_i^{i \in 1..n, n \geq 1} : \langle T_i^{i \in 1..n, n \geq 1}, \Gamma_i \rangle}{\Gamma_0 \vdash sml.id(p_i^{i \in 1..n, n \geq 1}) : \langle T, \Gamma_n \rangle} \qquad \texttt{(sml-call-decl)}$$

$$\frac{\begin{array}{c} id : (T_i^{i \in 1..n, n \geq 1}) \rightarrow T \notin \Gamma_0 \\ \Gamma_1 = \Gamma_0, `i, `m, id : (`i^{i \in 1..n, n \geq 1}) \rightarrow `m \\ \Gamma_i \vdash p_i^{i \in 1..n, n \geq 1} : \langle T_i^{i \in 1..n, n \geq 1}, \Gamma_{i+1} \rangle \\ \Gamma_{n+2} = \Gamma_{n+1}, `i : T_i^{i \in 1..n, n \geq 1} \end{array}}{\Gamma_0 \vdash sml.id(p_i^{i \in 1..n, n \geq 1}) : \langle `m, \Gamma_{n+2} \rangle} \qquad \texttt{(sml-call-infer)}$$

Figure 5.2: Analysis of SML function calls

Example The following examples illustrate the analysis of SML function calls given the following list of function signature declarations:

```
UserDefinedFunctions =
sig
    unit2Expr:  unit -> <expr>
    expr2Expr:  <expr> -> <expr>
end
```

- expr$_1$ → expr$_1$ if { sml.unit2Expr() } : type error

  because the context surrounding the function invocation expects a boolean value, but is given `expr[‘a]`

- expr$_1$ → expr$_1$ if { sml.unit2Expr() = expr$_1$ } :

  expr[‘a] → expr[‘a]

- expr$_1$ → expr$_1$ if { sml.undeclared01() } : expr[‘a] → expr[‘a]

- expr$_1$ → expr$_2$ if { expr$_2$ = sml.expr2Expr(expr$_1$) } :

  expr[‘a] → expr[‘b]

- expr$_1$ → id$_1$ if { id$_1$ = sml.expr2Expr(expr$_1$) } : type error because match equation's operands' types do not match

- expr$_1$ → expr$_1$ if { sml.foo(expr$_1$) andalso sml.foo() } : type error because after the first function call, the inferred type is expr[‘a] → ‘b, which does not match the second call's type unit → ‘c

- expr$_1$ → expr$_1$ if { id$_1$ = sml.foo(expr$_1$) andalso sml.foo(id$_1$) } : type error because expr[‘a] → id[‘b] does not match id[‘b] → ‘c

- expr$_1$ → expr$_1$ if { sml.foo(expr$_1$) orelse sml.foo() } : expr[‘a] → expr[‘a] because bindings produced from the analysis of one of orelse's operands are not remembered in the analysis of the second operand; the join type of expr[‘a] → ‘b and unit → ’c is ‘d

- id$_1$ → id$_1$ if { sml.foo(expr⟦id$_1$⟧) orelse sml.foo(expr⟦y⟧) } : id[‘a] → id[‘a] where the join type of expr[id[‘a]] → ‘b and expr[y] → ’c is expr[y] → ‘c

□

5.2 Primitive operations

In addition to access to SML functions to perform non-transformational tasks such as imperative side effects, *TL* provides built-in primitive operations on parse tree leaves. This provides a programmer the convenience of modifying terms directly within a transformation program without needing to define and call SML functions.

Figure 5.3 summarizes the primitive operations built into the language. All of the operators in *TL* are left-associative. The operator precedence, including the precedence of strategic operators, from lowest to highest is listed in Figure 5.4.

<table>
<tr><td>s</td><td>::=</td><td>...</td><td><i>strategies:</i></td></tr>
<tr><td></td><td></td><td>s binop s</td><td><i>binary operation</i></td></tr>
<tr><td></td><td></td><td>unop s</td><td><i>unary operation</i></td></tr>
<tr><td></td><td></td><td>boolVal</td><td><i>boolean value</i></td></tr>
<tr><td></td><td></td><td>intVal</td><td><i>integer value</i></td></tr>
<tr><td></td><td></td><td>realVal</td><td><i>real value</i></td></tr>
<tr><td></td><td></td><td>stringVal</td><td><i>string value</i></td></tr>
<tr><td>binop</td><td>::=</td><td>|| | &&</td><td><i>boolean ops</i></td></tr>
<tr><td></td><td></td><td>== | != | < | <= | > | >=</td><td><i>relational ops</i></td></tr>
<tr><td></td><td></td><td>+ | - | * | / | div | mod</td><td><i>arithmetic ops</i></td></tr>
<tr><td></td><td></td><td>ˆ</td><td><i>string concatenation</i></td></tr>
<tr><td>unop</td><td>::=</td><td>!</td><td><i>negation</i></td></tr>
<tr><td></td><td></td><td>˜</td><td><i>unary minus</i></td></tr>
</table>

Figure 5.3: Primitive operations in *TL*

<table>
<tr><td>orelse</td><td><i>match disjunction</i></td><td><i>lowest</i></td></tr>
<tr><td>andalso</td><td><i>match conjunction</i></td><td></td></tr>
<tr><td>=</td><td><i>match equation</i></td><td></td></tr>
<tr><td><+ , +> , <+></td><td><i>choice</i></td><td></td></tr>
<tr><td><; , ;></td><td><i>non-strict sequence</i></td><td></td></tr>
<tr><td><* , *></td><td><i>strict sequence</i></td><td></td></tr>
<tr><td>→</td><td><i>rule</i></td><td></td></tr>
<tr><td>if</td><td><i>rule condition</i></td><td></td></tr>
<tr><td>||</td><td><i>disjunction</i></td><td></td></tr>
<tr><td>&&</td><td><i>conjunction</i></td><td></td></tr>
<tr><td>==, !=, <, <=, >, >=</td><td><i>relational operators</i></td><td></td></tr>
<tr><td>ˆ, +, -</td><td><i>string concatenation, addition, subtraction</i></td><td></td></tr>
<tr><td>*, /, div, mod</td><td><i>multiply, real divide, integer divide, modulo</i></td><td></td></tr>
<tr><td>˜, !, not</td><td><i>unary minus, negation, match negation</i></td><td><i>highest</i></td></tr>
</table>

Figure 5.4: Precedence of operators in *TL*

Primitive operations are overloaded with respect to operands' types. For example, the acceptable type signatures for arithmetic addition are $int * int \rightarrow int$ as well as $real * real \rightarrow real$. In addition, since (parse) trees are ubiquitous in strategic rewriting, the operations are defined not only on operands of primitive type *prim* – $bool, int, real, string$ – but also on operands of tree types $t[prim]$, where the tree must be linear with a leaf of a primitive type. A *linear tree* is a tree, where each node has at most one child node.

Example The following examples illustrate primitive operations and their results

- $\text{expr}\llbracket 1 \rrbracket + 1 \equiv \text{expr}\llbracket 2 \rrbracket$

- $\text{expr}\llbracket 1 \rrbracket + \text{expr}\llbracket 2 \rrbracket \equiv \text{expr}\llbracket 3 \rrbracket$

- $\text{expr}\llbracket \text{Hello} \rrbracket$ ^“ ” ^$\text{expr}\llbracket \text{world} \rrbracket$ ^“!” $\equiv \text{expr}\llbracket \text{Hello world!} \rrbracket$

- $\text{expr}\llbracket 1 \rrbracket <= \text{expr}\llbracket 2 \rrbracket$ && $\text{expr}\llbracket 1.0 \rrbracket <= \text{expr}\llbracket 2.0 \rrbracket \equiv$ true

□

Figure 5.5 summarizes the typing rules for primitive values. In the base case, primitive values are assigned their corresponding types. Rule `(prim-term)` states that if, in the process of analyzing the primitive operations, we encounter a linear tree $t\llbracket leaf \rrbracket$, whose leaf is one of the primitive types T, then the term's type becomes more abstract as $t[T]$, instead of the usual type $t[leaf]$. Abstraction of term types is necessary because primitive values are already abstracted. For example, the type of term $(boolean, [(boolLex, [(true, [])])])$ becomes $(boolean, [(boolLex, [(bool, [])])])$. Note that we do not encode all of these details inside rule `(prim-term)` to avoid notational complexities.

Rule `(prim-b-or)` in Figure 5.6 analyzes boolean disjunction by calculating the operands' types and checking that they are either a boolean primitive type or a boolean tree type. If so, the return type is a *treeOf* the two types: i.e. if one or both types are tree types, then the result is the tree type $t[bool]$; otherwise, the result is the primitive type *bool*. Analysis of boolean conjunction and negation is similar.

Examples

- `true` $||$ `false :  bool`

- `true` $||$ `expr`$\llbracket$`false`$\rrbracket$ `:  expr[bool]`

- $\texttt{expr}[\![\texttt{expr}_1\ \texttt{+}\ \texttt{expr}_2]\!] \ \rightarrow\ \texttt{expr}_1 \mid\mid \texttt{expr}_2$:

  $\texttt{expr[bool + bool]} \ \rightarrow\ \texttt{expr[bool]}$

- $\texttt{true} \mid\mid \texttt{expr}[\![\texttt{"false"}]\!]$: $\texttt{type error}$ because $\texttt{"false"}$ is of type $\texttt{string}$

- $\texttt{true} \mid\mid \texttt{expr}[\![\texttt{(false)}]\!]$: $\texttt{type error}$ because $\texttt{expr}[\![\texttt{(false)}]\!]$ is non-linear

□

$$\frac{t \in \{true, false\}}{\Gamma \vdash t : \langle bool, \Gamma\rangle} \qquad \texttt{(prim-bool)}$$

$$\frac{t \in \mathbb{Z}}{\Gamma \vdash t : \langle int, \Gamma\rangle} \qquad \texttt{(prim-int)}$$

$$\frac{t \in \mathbb{R}}{\Gamma \vdash t : \langle real, \Gamma\rangle} \qquad \texttt{(prim-real)}$$

$$\frac{t \in \text{``}alphanum^{*}\text{''}}{\Gamma \vdash t : \langle string, \Gamma\rangle} \qquad \texttt{(prim-string)}$$

$$\frac{\Gamma \vdash leaf : \langle T, \Gamma_1\rangle \qquad T \in \{bool, int, real, string\}}{\Gamma \vdash t[\![leaf]\!] : \langle t[T], \Gamma_1\rangle} \qquad \texttt{(prim-term)}$$

Figure 5.5: Analysis of primitive values

$$\frac{\Gamma \vdash s_1 : \langle T_1, \Gamma_1\rangle \qquad \Gamma_1 \vdash s_2 : \langle T_2, \Gamma_2\rangle \qquad T_1, T_2 \in \{bool, t[bool]\}}{\Gamma \vdash s_1 \mid\mid s_2 : \langle treeOf(T_1, T_2), \Gamma_2\rangle} \qquad \texttt{(prim-b-or)}$$

$$\frac{\Gamma \vdash s_1 : \langle T_1, \Gamma_1\rangle \qquad \Gamma_1 \vdash s_2 : \langle T_2, \Gamma_2\rangle \qquad T_1, T_2 \in \{bool, t[bool]\}}{\Gamma \vdash s_1 \ \&\&\ s_2 : \langle treeOf(T_1, T_2), \Gamma_2\rangle} \qquad \texttt{(prim-b-and)}$$

$$\frac{\Gamma \vdash s : \langle T, \Gamma_1\rangle \qquad T \in \{bool, t[bool]\}}{\Gamma \vdash !\ s : \langle T, \Gamma_1\rangle} \qquad \texttt{(prim-b-not)}$$

Figure 5.6: Analysis of boolean operations

Analysis of relational operations, summarized in Figure 5.7, is similar to the analysis of boolean operations except that relational operations are overloaded and the acceptable types of operands are all four primitive types in the case of equality and

$$\frac{\begin{array}{c}\Gamma \vdash s_1 : \langle T_1, \Gamma_1 \rangle \qquad \Gamma_1 \vdash s_2 : \langle T_2, \Gamma_2 \rangle \\ (T_1, T_2 \in \{\mathit{bool}, t[\mathit{bool}]\} \ \mathit{or} \ T_1, T_2 \in \{\mathit{int}, t[\mathit{int}]\} \ \mathit{or} \\ T_1, T_2 \in \{\mathit{real}, t[\mathit{real}]\} \ \mathit{or} \ T_1, T_2 \in \{\mathit{string}, t[\mathit{string}]\})\end{array}}{\Gamma \vdash s_1 \ == \ s_2 : \langle \mathit{bool}, \Gamma_2 \rangle} \qquad \texttt{(prim-eq)}$$

$$\frac{\begin{array}{c}\Gamma \vdash s_1 : \langle T_1, \Gamma_1 \rangle \qquad \Gamma_1 \vdash s_2 : \langle T_2, \Gamma_2 \rangle \\ (T_1, T_2 \in \{\mathit{bool}, t[\mathit{bool}]\} \ \mathit{or} \ T_1, T_2 \in \{\mathit{int}, t[\mathit{int}]\} \ \mathit{or} \\ T_1, T_2 \in \{\mathit{real}, t[\mathit{real}]\} \ \mathit{or} \ T_1, T_2 \in \{\mathit{string}, t[\mathit{string}]\})\end{array}}{\Gamma \vdash s_1 \ != \ s_2 : \langle \mathit{bool}, \Gamma_2 \rangle} \qquad \texttt{(prim-neq)}$$

$$\frac{\begin{array}{c}\Gamma \vdash s_1 : \langle T_1, \Gamma_1 \rangle \qquad \Gamma_1 \vdash s_2 : \langle T_2, \Gamma_2 \rangle \\ (T_1, T_2 \in \{\mathit{int}, t[\mathit{int}]\} \ \mathit{or} \ T_1, T_2 \in \{\mathit{real}, t[\mathit{real}]\} \ \mathit{or} \ T_1, T_2 \in \{\mathit{string}, t[\mathit{string}]\})\end{array}}{\Gamma \vdash s_1 \ < \ s_2 : \langle \mathit{bool}, \Gamma_2 \rangle} \qquad \texttt{(prim-lt)}$$

$$\frac{\begin{array}{c}\Gamma \vdash s_1 : \langle T_1, \Gamma_1 \rangle \qquad \Gamma_1 \vdash s_2 : \langle T_2, \Gamma_2 \rangle \\ (T_1, T_2 \in \{\mathit{int}, t[\mathit{int}]\} \ \mathit{or} \ T_1, T_2 \in \{\mathit{real}, t[\mathit{real}]\} \ \mathit{or} \ T_1, T_2 \in \{\mathit{string}, t[\mathit{string}]\})\end{array}}{\Gamma \vdash s_1 \ <= \ s_2 : \langle \mathit{bool}, \Gamma_2 \rangle} \qquad \texttt{(prim-leq)}$$

$$\frac{\begin{array}{c}\Gamma \vdash s_1 : \langle T_1, \Gamma_1 \rangle \qquad \Gamma_1 \vdash s_2 : \langle T_2, \Gamma_2 \rangle \\ (T_1, T_2 \in \{\mathit{int}, t[\mathit{int}]\} \ \mathit{or} \ T_1, T_2 \in \{\mathit{real}, t[\mathit{real}]\} \ \mathit{or} \ T_1, T_2 \in \{\mathit{string}, t[\mathit{string}]\})\end{array}}{\Gamma \vdash s_1 \ > \ s_2 : \langle \mathit{bool}, \Gamma_2 \rangle} \qquad \texttt{(prim-gt)}$$

$$\frac{\begin{array}{c}\Gamma \vdash s_1 : \langle T_1, \Gamma_1 \rangle \qquad \Gamma_1 \vdash s_2 : \langle T_2, \Gamma_2 \rangle \\ (T_1, T_2 \in \{\mathit{int}, t[\mathit{int}]\} \ \mathit{or} \ T_1, T_2 \in \{\mathit{real}, t[\mathit{real}]\} \ \mathit{or} \ T_1, T_2 \in \{\mathit{string}, t[\mathit{string}]\})\end{array}}{\Gamma \vdash s_1 \ >= \ s_2 : \langle \mathit{bool}, \Gamma_2 \rangle} \qquad \texttt{(prim-geq)}$$

Figure 5.7: Analysis of relational operations

non-equality operators (rules `(prim-eq)` and `(prim-neq)`) and primitive types *int*, *real* or *string* for the remaining four relational operators.

Figure 5.8 summarizes typing rules for arithmetic operations. Addition, subtraction and multiplication are overloaded operations, while real division, integer division, modulo, string concatenation and unary minus operations are non-overloaded operations.

Examples

- `~1 :  int`
- `~1.0 :  real`
- `expr`$_1$ $\rightarrow$ `~expr`$_1$ `:  expr[int]`

$$\frac{\begin{array}{c}\Gamma \vdash s_1 : \langle T_1, \Gamma_1 \rangle \qquad \Gamma_1 \vdash s_2 : \langle T_2, \Gamma_2 \rangle \\ (T_1, T_2 \in \{int, t[int]\}\ or\ T_1, T_2 \in \{real, t[real]\})\end{array}}{\Gamma \vdash s_1\ +\ s_2 : \langle treeOf(T_1, T_2), \Gamma_2 \rangle} \qquad \texttt{(prim-plus)}$$

$$\frac{\begin{array}{c}\Gamma \vdash s_1 : \langle T_1, \Gamma_1 \rangle \qquad \Gamma_1 \vdash s_2 : \langle T_2, \Gamma_2 \rangle \\ (T_1, T_2 \in \{int, t[int]\}\ or\ T_1, T_2 \in \{real, t[real]\})\end{array}}{\Gamma \vdash s_1\ -\ s_2 : \langle treeOf(T_1, T_2), \Gamma_2 \rangle} \qquad \texttt{(prim-minus)}$$

$$\frac{\begin{array}{c}\Gamma \vdash s_1 : \langle T_1, \Gamma_1 \rangle \qquad \Gamma_1 \vdash s_2 : \langle T_2, \Gamma_2 \rangle \\ (T_1, T_2 \in \{int, t[int]\}\ or\ T_1, T_2 \in \{real, t[real]\})\end{array}}{\Gamma \vdash s_1\ *\ s_2 : \langle treeOf(T_1, T_2), \Gamma_2 \rangle} \qquad \texttt{(prim-times)}$$

$$\frac{\Gamma \vdash s_1 : \langle T_1, \Gamma_1 \rangle \qquad \Gamma_1 \vdash s_2 : \langle T_2, \Gamma_2 \rangle \qquad T_1, T_2 \in \{real, t[real]\}}{\Gamma \vdash s_1\ /\ s_2 : \langle treeOf(T_1, T_2), \Gamma_2 \rangle} \qquad \texttt{(prim-divide)}$$

$$\frac{\Gamma \vdash s_1 : \langle T_1, \Gamma_1 \rangle \qquad \Gamma_1 \vdash s_2 : \langle T_2, \Gamma_2 \rangle \qquad T_1, T_2 \in \{int, t[int]\}}{\Gamma \vdash s_1\ div\ s_2 : \langle treeOf(T_1, T_2), \Gamma_2 \rangle} \qquad \texttt{(prim-div)}$$

$$\frac{\Gamma \vdash s_1 : \langle T_1, \Gamma_1 \rangle \qquad \Gamma_1 \vdash s_2 : \langle T_2, \Gamma_2 \rangle \qquad T_1, T_2 \in \{int, t[int]\}}{\Gamma \vdash s_1\ mod\ s_2 : \langle treeOf(T_1, T_2), \Gamma_2 \rangle} \qquad \texttt{(prim-mod)}$$

$$\frac{\Gamma \vdash s_1 : \langle T_1, \Gamma_1 \rangle \qquad \Gamma_1 \vdash s_2 : \langle T_2, \Gamma_2 \rangle \qquad T_1, T_2 \in \{string, t[string]\}}{\Gamma \vdash s_1\ \hat{}\ s_2 : \langle treeOf(T_1, T_2), \Gamma_2 \rangle} \qquad \texttt{(prim-concat)}$$

$$\frac{\Gamma \vdash s : \langle T, \Gamma_1 \rangle \qquad (T \in \{int, t[int]\}\ or\ T \in \{real, t[real]\})}{\Gamma \vdash \tilde{}s : \langle T, \Gamma_1 \rangle} \qquad \texttt{(prim-tilde)}$$

Figure 5.8: Analysis of arithmetic operations

- $\texttt{expr}_1 \ \rightarrow\ \texttt{expr}_1\ \texttt{+ 1 :}\ \ \texttt{expr[int]}$

- `expr⟦2⟧ + expr⟦2.2⟧ :` `type error` because both operands must be of the same type

□

5.3 Transient strategies

Transient strategies and related operators *opaque*, *raise*, *hide* and *lift* modify the behavior of a strategy during execution. For example, successful application of a strategy to a term may reduce it to the no-op *SKIP* and prevent other strategies in conditional composition from applying to a term: i.e. $transient(s_1) \mathbin{<\!\!\!+} s_2$. Because the type system performs its analysis statically, the analysis needs to be conservatively approximating the conditions at run-time. Therefore, these operators do not modify the type of their strategy, which is a conservative assumption that a strategy may or may never apply. In other words, given $s_1 : T_{11} \to T_{12}$ and $s_2 : T_{21} \to T_{22}$ the type of a strategy $transient(s_1) \mathbin{<\!\!\!+} s_2$ is $T_{11} \to T_{12} + T_{21} \to T_{22}$ even if s_1 happens to always apply at run-time and the entire composition becomes just s_2 after the first application. Figure 5.9 summarizes the analysis of transient strategies and related operators.

$$\frac{\Gamma \vdash s : \langle T, \Gamma_1 \rangle}{\Gamma \vdash transient(s) : \langle T, \Gamma_1 \rangle} \qquad \texttt{(transient)}$$

$$\frac{\Gamma \vdash s : \langle T, \Gamma_1 \rangle}{\Gamma \vdash opaque(s) : \langle T, \Gamma_1 \rangle} \qquad \texttt{(opaque)}$$

$$\frac{\Gamma \vdash s : \langle T, \Gamma_1 \rangle}{\Gamma \vdash raise(s) : \langle T, \Gamma_1 \rangle} \qquad \texttt{(raise)}$$

$$\frac{\Gamma \vdash s : \langle T, \Gamma_1 \rangle}{\Gamma \vdash hide(s) : \langle T, \Gamma_1 \rangle} \qquad \texttt{(hide)}$$

$$\frac{\Gamma \vdash s : \langle T, \Gamma_1 \rangle}{\Gamma \vdash lift(s) : \langle T, \Gamma_1 \rangle} \qquad \texttt{(lift)}$$

Figure 5.9: Analysis of transient strategies

5.4 Local recursive strategies

Conditional rewrite rules in *TL* may contain definitions of recursive strategies of the form $id := s$, where s may be an arbitrary strategy with recursive references to id. Figure 5.10 summarizes the analysis of local recursive strategies. In particular, prior to the analysis of the strategy, the typing context is extended by binding the identifier id to a fresh type variable. This enables analysis of recursive references. Upon completion of the analysis, the existing binding is updated with the calculated type T. Because the context of local recursive strategy definitions is a match expression (i.e. a rule's condition), the resulting type is *bool*.

$$\frac{\Gamma_1 = \Gamma, \text{‘}a, id : \text{‘}a \qquad \Gamma_1 \vdash s : \langle T, \Gamma_2 \rangle \qquad \Gamma_3 = \Gamma_2, id : T}{\Gamma \vdash id := s : \langle bool, \Gamma_3 \rangle} \qquad \texttt{(bind)}$$

Figure 5.10: Analysis of local recursive strategies

Example The following example illustrates the use and analysis of local recursive strategies:

```
inc:  expr1  → expr2 // increments all expressions except expr⟦999⟧
      if { expr2 = expr1 + 1 andalso
           not(expr2 = expr⟦1000⟧)
      }
// type of inc is expr[int]  → expr[int]


fix:  expr1  → expr2 // counts up until expr⟦999⟧
      if { s := inc  <* (s  <+ ID) andalso
           expr2 = s expr1
      }
// type of fix is expr[int]  → expr[int]
```

□

5.5 Higher-order rules and compositions

TL supports higher-order rewrite rules. If there is more than one dynamic rule generated by application of a higher-order rule, then the new rules can be composed using operator *fold*. Analysis of strategic expressions involving *fold* needs to ensure that composition of dynamic rules does not contain any errors: e.g. the composition needs to have a chance to succeed. For example, the strategy

```
fold  <*  ( expr⟦a⟧  → expr⟦b⟧  → expr⟦c⟧  <*
            expr⟦a⟧  → expr⟦y⟧  → expr⟦z⟧
          )
```

contains an error, because the dynamically generated composition

```
expr⟦b⟧ → expr⟦c⟧  <*  expr⟦y⟧ → expr⟦z⟧
```

will always fail for all inputs. Thus, the goal of analyzing higher-order rules and compositions is to flag as many errors as can be determined statically.

Figure 5.11 summarizes the analysis of *fold* expressions. Upon calculation of the type of the argument strategy s, further analysis is performed by function $\mathit{foldRules}_{\oplus}$, where $\oplus$ is the binary combinator used in the *fold* expression.

$$\frac{\Gamma \vdash s : \langle T_1, \Gamma_1 \rangle \qquad \Gamma_1 \vdash \mathit{foldRules}_{\oplus}(T_1) : \langle T_2, \Gamma_2 \rangle}{\Gamma \vdash \mathit{fold} \oplus s : \langle T_2, \Gamma_2 \rangle} \quad \texttt{(fold)}$$

Figure 5.11: Analysis of higher-order compositions

Since the strategy s may be a composite strategy, we augment the previous analysis of compositions by extending functions *ckStar* and *ckSeq*, which were previously defined in Figures 4.11 and 4.13. Figure 5.12 summarizes the extension. In particular, rule (`ckstar-hi-rule`$^+$) states that composition of two higher-order rules is well-typed if the outer input patterns of the two rules are type-unifiable. If so, a special-purpose rule type $T \to [T_i^{i \in 1..n}]$ is returned to $\mathit{foldRules}_{\oplus}$ to analyze the composition $T_1 \oplus T_2 \oplus \cdots \oplus T_n$. Otherwise, the composition is a type error, which is indicated by $\emptyset$.

Similar analysis applies to non-strict sequential composition checked by rule (ckseq-hi-rule$^+$). However, here there are three different execution paths through non-strict sequence $a \mathbin{<;} b$: they are a and b, just a and just b. Therefore, the set of return types includes three possible rule types in case of $T_a = T_x$, and two possible rule types otherwise.

$$\frac{T_a = T_x}{\begin{array}{c}\Gamma \vdash \mathit{ckStar}(T_a \to T_b \to T_c, T_x \to T_y \to T_z) : \\ \langle \{T_a \to [T_b \to T_c, T_y \to T_z]\}, \Gamma \rangle\end{array}}$$

(ckstar-hi-rule$^+$)

$$\frac{T_a \neq T_x}{\Gamma \vdash \mathit{ckStar}(T_a \to T_b \to T_c, T_x \to T_y \to T_z) : \langle \emptyset, \Gamma \rangle}$$

(ckstar-hi-rule$^-$)

$$\frac{T_a = T_x}{\begin{array}{c}\Gamma \vdash \mathit{ckSeq}(T_a \to T_b \to T_c, T_x \to T_y \to T_z) : \\ \langle \{T_a \to [T_b \to T_c, T_y \to T_z], T_a \to T_b \to T_c, T_x \to T_y \to T_z\}, \Gamma \rangle\end{array}}$$

(ckseq-hi-rule$^+$)

$$\frac{T_a \neq T_x \qquad \mathit{canReach}(T_a \to T_b \to T_c, T_x \to T_y \to T_z)}{\begin{array}{c}\Gamma \vdash \mathit{ckSeq}(T_a \to T_b \to T_c, T_x \to T_y \to T_z) : \\ \langle \{T_a \to T_b \to T_c, T_x \to T_y \to T_z\}, \Gamma \rangle\end{array}}$$

(ckseq-hi-rule$^-$)

Figure 5.12: Extension: higher-order compositions

Finally, Figure 5.13 specifies the definition of $foldRules$. Here, if the argument is a single higher-order rule, then there is nothing to fold and the rule is returned as the result – rule (fold-rules-base). If the argument is a sum type, then each of its operands are analyzed recursively and the results are combined into a sum type – rule (fold-rules-sum). Rule (fold-rules- list) states that if the argument is a rule type, where the right-hand side is a list of types, then the list of types is converted into a single type by function $createComp$.

Figure 5.14 summarizes the definition of $createComp$. If the combinator is one of the conditional combinators , <+> , <+ , +> , <* , *> , <; , ;> , and <+> and <+ and +> and <* and *> and <; and ;> then the resulting type is a sum of list elements. Otherwise, the list elements are passed onto functions *ckStar* and *ckSeq* to obtain the set of possible types. For notational simplicity, we implicitly lift binary compositions

$$\frac{}{\Gamma \vdash \mathit{foldRules}_{\oplus}(T_a \to T_b \to T_c) : \langle T_a \to T_b \to T_c, \Gamma\rangle} \qquad \texttt{(fold-rules-base)}$$

$$\frac{\Gamma \vdash \mathit{foldRules}_{\oplus}(T_a) : \langle T'_a, \Gamma_1\rangle \qquad \Gamma \vdash \mathit{foldRules}_{\oplus}(T_b) : \langle T'_b, \Gamma_2\rangle}{\Gamma \vdash \mathit{foldRules}_{\oplus}(T_a + T_b) : \langle T'_a + T'_b, \Gamma\rangle} \qquad \texttt{(fold-rules-sum)}$$

$$\frac{\Gamma \vdash \mathit{createComp}_{\oplus}[T_i^{i\in 1..n, n\geq 2}] : \langle T_{is}, \Gamma_1\rangle}{\Gamma \vdash \mathit{foldRules}_{\oplus}(T_a \to [T_i^{i\in 1..n, n\geq 2}]) : \langle T_a \to T_{is}, \Gamma_2\rangle} \qquad \texttt{(fold-rules-list)}$$

Figure 5.13: Folding dynamic rules

performed by the two functions to n-ary compositions. The elements of the returned set are then combined into a sum type.

$$\frac{}{\Gamma \vdash \mathit{createComp}_{<\!+\!>}[T_i^{i\in 1..n, n\geq 2}] : \langle T_1 + T_2 + \cdots + T_n, \Gamma\rangle} \qquad \texttt{(create-comp-acond)}$$

$$\frac{}{\Gamma \vdash \mathit{createComp}_{<\!+}[T_i^{i\in 1..n, n\geq 2}] : \langle T_1 + T_2 + \cdots + T_n, \Gamma\rangle} \qquad \texttt{(create-comp-lcond)}$$

$$\frac{}{\Gamma \vdash \mathit{createComp}_{+\!>}[T_i^{i\in 1..n, n\geq 2}] : \langle T_n + T_n + \cdots + T_1, \Gamma\rangle} \qquad \texttt{(create-comp-rcond)}$$

$$\frac{\Gamma \vdash \mathit{ckStar}[T_i^{i\in 1..n, n\geq 2}] : \langle \{S_j^{j\in 1..m, m\geq 1}\}, \Gamma_1\rangle}{\Gamma \vdash \mathit{createComp}_{<\!*}[T_i^{i\in 1..n, n\geq 2}] : \langle S_1 + S_2 + \cdots + S_m, \Gamma_1\rangle} \qquad \texttt{(create-comp-lstar)}$$

$$\frac{\Gamma \vdash \mathit{ckStar}[T_i^{i\in n..1, n\geq 2}] : \langle \{S_j^{j\in 1..m, m\geq 1}\}, \Gamma_1\rangle}{\Gamma \vdash \mathit{createComp}_{*\!>}[T_i^{i\in 1..n, n\geq 2}] : \langle S_1 + S_2 + \cdots + S_m, \Gamma_1\rangle} \qquad \texttt{(create-comp-rstar)}$$

$$\frac{\Gamma \vdash \mathit{ckSeq}[T_i^{i\in 1..n, n\geq 2}] : \langle \{S_j^{j\in 1..m, m\geq 1}\}, \Gamma_1\rangle}{\Gamma \vdash \mathit{createComp}_{<;}[T_i^{i\in 1..n, n\geq 2}] : \langle S_1 + S_2 + \cdots + S_m, \Gamma_1\rangle} \qquad \texttt{(create-comp-lseq)}$$

$$\frac{\Gamma \vdash \mathit{ckSeq}[T_i^{i\in n..1, n\geq 2}] : \langle \{S_j^{j\in 1..m, m\geq 1}\}, \Gamma_1\rangle}{\Gamma \vdash \mathit{createComp}_{;>}[T_i^{i\in 1..n, n\geq 2}] : \langle S_1 + S_2 + \cdots + S_m, \Gamma_1\rangle} \qquad \texttt{(create-comp-rseq)}$$

Figure 5.14: Composing dynamic rules

Example Assuming the following definitions, the examples below demonstrate the results of type analysis:

abc: $\texttt{expr}[\![\mathsf{a}]\!] \to \texttt{expr}[\![\mathsf{b}]\!] \to \texttt{expr}[\![\mathsf{c}]\!]$

ayz: $\texttt{expr}[\![\mathsf{a}]\!] \to \texttt{expr}[\![\mathsf{y}]\!] \to \texttt{expr}[\![\mathsf{z}]\!]$

xyz: expr⟦x⟧ → expr⟦y⟧ → expr⟦z⟧

- fold <+ *ID* : type error because *ID* is not a higher-order rule
- (abc <* ayz) expr⟦a⟧: type error because a higher-order strategy is applied without an enclosing fold
- fold <+ abc: expr[a] → expr[b] → expr[c]
- fold <+ abc expr⟦a⟧: expr[b] → expr[c]
- fold <+ (abc <* ayz):
  expr[a] → (expr[b] → expr[c] + expr[y] → expr[z])
- fold <+ (abc <* ayz) id⟦a⟧: type error because none of the higher-order rule's patterns match the input term
- fold <+ (abc <* ayz) expr⟦a⟧ expr⟦y⟧: expr[z]
- fold <* (abc <* ayz): type error because dynamic rules cannot be composed using <*
- fold <+ (abc <* xyz): type error because higher-order rules' input patterns do not match
- fold <+ (abc <+ xyz):
  expr[a] → expr[b] → expr[c] + expr[x] → expr[y] → expr[z]

□

5.6 Traversals

TL provides generic one-layer iterators *mapL*, *mapR* and *mapB* that apply their argument strategy to immediate sub-terms of a term. Given a traversal strategy *mapL s*, it is an error to apply it to a term, whose immediate sub-terms are not applicable for *s*. In such case, the traversal strategy can never succeed. Therefore, the goal of type analysis is to identify such conditions and flag them as type errors.

Figure 5.15 summarizes analysis of one-layer traversals. Note that the semantic differences of the three kinds of traversals are only manifested if the argument strategy contains transient strategies (see Section 3.6.2 for the discussion). Because static type analysis does not differentiate between transient and non-transient strategies (Section 5.3), the analysis of one-layer traversals does not distinguish between the three kinds of operators as well. Therefore, rule `(one-layer-trav)` calculates the type of the argument strategy *s* and assigns type *map T* to all three kinds of traversals.

$$\frac{\mathit{travOne} \in \{\mathit{mapL}, \mathit{mapR}, \mathit{mapB}\} \qquad \Gamma \vdash s : \langle T, \Gamma_1 \rangle}{\Gamma \vdash \mathit{travOne}\ s : \langle \mathit{map}\ T, \Gamma_1 \rangle} \quad \texttt{(one-layer-trav)}$$

Figure 5.15: Analysis of one-layer traversals

Due to the introduction of a new constructor *map T*, our goal in this section is to extend existing analysis to account for the new constructor. In particular, to detect ill-typed applications we need to extend the analysis of application.

Figure 5.16 summarizes the extension of the analysis of application. In the base case, indicated by rule `(red-map-leaf)`, application of one-layer traversal to a term with no sub-terms (leaf) is well-typed, because there are no sub-terms to apply to. Otherwise, as expressed by rule `(red-map-term)` if the argument strategy is rewrite rule $T_a \rightarrow T_b$ and the term has one or more sub-terms $(T, [T_i^{i \in 1..n, n \geq 1}])$, then the argument rule is applied to each of the sub-terms to obtain sets of possible types S_i. Then, for each of the possible new sub-terms a new term is created such that each of the sub-terms is represented in at least one new term. Then, the list of possible new terms is filtered to ensure that only terms with valid (parse-able) structure remain. Finally, all of the valid possible new terms are combined into a sum type, which becomes the final result of one-layer traversal application. If all applications are ill-typed or if all applications produce invalid terms, then a type error is raised by the analysis.

Example Given the following definitions, the examples below illustrate the analysis of one-layer traversal application

sum: expr⟦x + y⟧

prim: expr$_1$ → expr⟦b⟧ if { expr$_1$ = expr⟦1⟧ + 1 }

- mapL (expr$_1$ → expr⟦2⟧) sum : expr[2 + 2]
- mapL (expr⟦x⟧ → expr⟦a + b⟧) sum : expr[a + b + y]
- mapL (id⟦x⟧ → id⟦z⟧) sum : type error because there does not exist an immediate sub-term id⟦x⟧
- mapL (expr⟦a⟧ → expr⟦z⟧) sum : type error because neither expr⟦x⟧ nor expr⟦y⟧ match expr⟦a⟧
- mapL (expr⟦x⟧ → id⟦z⟧) sum : type error because the resulting term $(\mathit{expr}, [(\mathit{id}, \cdots), (+, [\ ]), (\mathit{expr}, \cdots)])$ does not have a valid structure: i.e. it is not parse-able with the grammar of language *Expr* (see Figure 3.2)
- mapL prim expr⟦2 + c⟧: expr[b + c] + expr[2 + c]

□

Continuing with typing rule in Figure 5.16, rule (red-map-sum) states that analysis of one-layer traversal with a sum strategy type proceeds by applying each of the sum's operands to the input term and combining the resulting types into a union. The resulting environment is the extended intersection $\Gamma_1 \hat{\cap} \Gamma_2$, where a variable bound to two distinct types is bound to their join type (see Section 4.7 for the discussion of join types).

$$\frac{}{\Gamma \vdash \mathit{reduce}(\mathit{map}\ T, (t, [\,])) : \langle (t, [\,]), \Gamma \rangle} \qquad \texttt{(red-map-leaf)}$$

$$\frac{\forall i \in 1..n. \Gamma_{i-1} \vdash \mathit{reduce}(T_a \rightarrow T_b, T_i) : \langle S_i, \Gamma_i \rangle}{\Gamma_0 \vdash \mathit{reduce}(\mathit{map}\ T_a \rightarrow T_b, (T, [T_i^{i \in 1..n, n \geq 1}])) : \langle \{(T, [S_i^{i \in 1..n, n \geq 1}])\}, \Gamma_n \rangle} \qquad \texttt{(red-map-term)}$$

$$\frac{\Gamma \vdash \mathit{reduce}(T_a, T_c) : \langle S_{ac}, \Gamma_1 \rangle \qquad \Gamma \vdash \mathit{reduce}(T_b, T_c) : \langle S_{bc}, \Gamma_2 \rangle \quad \Gamma_1 \hat{\cap} \Gamma_2 = \Gamma_{12}}{\Gamma \vdash \mathit{reduce}(\mathit{map}(T_a + T_b), T_c) : \langle S_{ac} \cup S_{bc}, \Gamma_{12} \rangle} \qquad \texttt{(red-map-sum)}$$

Figure 5.16: Extension: application of one-layer traversals

Example Given the following definitions, the following examples illustrate the analysis and its results

ab: expr⟦a⟧ → expr⟦b⟧

bc: expr⟦b⟧ → expr⟦c⟧

- mapL (ab <+ bc) expr⟦a - b⟧ : expr[b - b] + expr[a - c]

- mapL (ab <* bc) expr⟦a - b⟧ : expr[c - b]

□

Chapter 6

An ML Implementation of Type Analysis

In the previous two chapters, we have presented the type analysis of rewrite strategies from the formal, yet abstract, perspective. It is often easier to work with a formal presentation, when it is grounded in a concrete implementation. Here, we describe the key components of the implementation of type analysis. The code presented here is written in Standard ML[1] [56][57][55][65]. However, this can be easily ported to SML's cousin in the ML family of languages [25] – Objective Caml [46][17], or other functional languages such as Haskell [31][63] or Scheme with pattern-matching extensions [36][21].

6.1 Syntax

Type analysis is a syntax-directed method of computing types based on the syntactic constructs of the language. Therefore, we first present the abstract syntax definition of *TL* . Figures 6.1 and 6.2 summarize these constructs. Each construct is listed with

[1] The code is available from Subversion repository at http://code.google.com/p/tl-types/. *TL* 's IDE – HATS is available from http://faculty.ist.unomaha.edu/winter/

a concrete syntax example on the right to provide an intuitive connection between abstract and concrete syntax.

Each node is decorated with a value of type `info`, which contains file position of the node including file name, line and column numbers. This information is created by the *TL* parser and is used by the type checker in reporting of type errors to assist the programmer in locating the source of an error.

6.2 Typing Context

Typing context tracks bindings of identifiers to types. For efficiency we use an ordered map data structure provided by SML/NJ libraries. Further, the map uses a binary search tree for a quick lookup of integer keys. The associated implementation in the library is `IntBinaryMap`:

```
structure S = IntBinaryMap
type context = ty S.map
val initialEnv: context = S.empty
```

To avoid inefficiencies of strings as much as possible, we hash identifiers and map hashed values to an integer counter. `HashTable` is an SML/NJ data structure.

```
exception SymbolExn
val hashtable: (string, int) HashTable.hash_table
  = HashTable.mkTable(HashString.hashString, op =) (128, SymbolExn)

val nextSym = ref 0
(* Looks up a symbol for a given name or creates a new one if none exists.
 * symbol: string -> (string, int) *)
fun symbol id = ...
```

Given these basic definitions, we can now invoke functions provided by ordered maps [61]. For example:

```
datatype expr =
    Bool         of bool * info                        true
|   Int          of int * info                         1
|   Real         of string * info                      1.0
|   String       of string * info                      “s”
|   Identifer    of string * info                      x

|   Concat       of expr * expr * info                 xˆy
|   Plus         of expr * expr * info                 x + y
|   Minus        of expr * expr * info                 x – y
|   Times        of expr * expr * info                 x * y
|   Divide       of expr * expr * info                 x / y
|   Div          of expr * expr * info                 x div y
|   Mod          of expr * expr * info                 x mod y
|   Tilde        of expr * info                        ˜x

|   Eq           of expr * expr * info                 x == y
|   Neq          of expr * expr * info                 x != y
|   Lt           of expr * expr * info                 x < y
|   Leq          of expr * expr * info                 x <= y
|   Gt           of expr * expr * info                 x > y
|   Geq          of expr * expr * info                 x >= y

|   BOr          of expr * expr * info                 x && y
|   BAnd         of expr * expr * info                 x || y
|   BNot         of expr * expr * info                 ! x

|   Sml0         of expr * info                        sml.f()
|   Sml          of expr * expr list * info            sml.f(args+)

|   Term         of itree * info                       expr⟦expr1 + x⟧
|   Bind         of expr * expr * info                 x := s
|   Match        of expr * expr * info                 expr⟦id1⟧ = expr⟦x⟧
|   Andalso      of expr * expr * info                 m andalso m
|   Orelse       of expr * expr * info                 m orelse m
|   Not          of expr * info                        not m

|   Rule         of expr * expr * info                 p → s
|   CRule        of expr * expr * expr * info          p → s if { m }
|   Id           of info                               ID
|   Skip         of info                               SKIP
|   App          of expr * expr * info                 s t

|   Transient    of expr * info                        transient(s)
|   Opaque       of expr * info                        opaque(s)
|   Raise        of expr * info                        raise(s)
|   Hide         of expr * info                        hide(s)
|   Lift         of expr * info                        lift(s)
```

Figure 6.1: Abstract syntax of *TL*

```
|  Choice          of expr * expr * info                r <+> s
|  LChoice         of expr * expr * info                r <+ s
|  RChoice         of expr * expr * info                r +> s
|  LStar           of expr * expr * info                r <* s
|  RStar           of expr * expr * info                r *> s
|  LSeq            of expr * expr * info                r <; s
|  RSeq            of expr * expr * info                r ;> s

|  FoldChoice      of expr * info                       fold <+> s
|  FoldLChoice     of expr * info                       fold <+ s
|  FoldRChoice     of expr * info                       fold +> s
|  FoldLStar       of expr * info                       fold <* s
|  FoldRStar       of expr * info                       fold *> s
|  FoldLSeq        of expr * info                       fold <; s
|  FoldRSeq        of expr * info                       fold ;> s

|  FoldSChoice     of expr * info                       foldS <+> s
|  FoldSLChoice    of expr * info                       foldS <+ s
|  FoldSRChoice    of expr * info                       foldS +> s
|  FoldSLStar      of expr * info                       foldS <* s
|  FoldSRStar      of expr * info                       foldS *> s
|  FoldSLSeq       of expr * info                       foldS <; s
|  FoldSRSeq       of expr * info                       foldS ;> s

|  MapL            of expr * info                       mapL s
|  MapR            of expr * info                       mapR s
|  MapB            of expr * info                       mapB s

|  Iterator        of expr * expr list * info           TDL s

|  Signature       of expr * info                       signatures
|  List            of expr list * info                  foo: bool → int

|  NonRecursive    of expr * expr * info                x: s
|  Recursive       of expr * expr list * expr * info    def x+ = s
|  Prog            of expr list * info;                 d+

datatype itree =
   ITree           of inode * itree list;               (x, [ ])

datatype inode =
   INode           of string * info                     x
|  IMatchVar       of string * string * info;           id1 or ⟨id⟩_1
```

Figure 6.2: Abstract syntax of *TL* (Cont'd)

```
S.insert(ctx1, #2(symbol id1), ty1)

S.find(ctx1, #2(symbol id1))

S.inDomain(ctx1, #2(symbol id1))
```

6.3 Types

The set of valid types is defined by the data type summarized in Figure 6.3.

```
datatype ty =
    TyBool | TyInt | TyReal | TyString                  primitive types
|   TyUnit                                              for SML function calls
|   TyVar of int                                        type variable ‘a
|   TyTerm of string * ty list                          type of a term
|   TyRule of ty * ty                                   arrow type
|   TySum of ty * ty                                    union type

|   TyList of ty list                                   for accumulation of dynamic rules
|   TyMap of ty                                         one-layer iterators
|   TyIter of string * ty                               full-term iterators

|   TyInf                                               type of TL constructs unsupported
|   TyError                                             for ill-typed declarations
                                                           at a given precision level
```

Figure 6.3: Types used by the analysis

In addition to the data type definition, utility functions provide auxiliary assistance. Their signatures are summarized below:

```
exception TypeError of string;


(* Applies a function to an argument in a bottom-up manner.
 * mapTy: (ty -> ty) -> ty -> ty *)
fun mapTy f (TyTerm (st1,ts)    ) = f (TyTerm (st1, map (mapTy f) ts))
...


(* Applies a function to an argument in a top-down manner.
 * mapTyTD: (ty -> ty) -> ty -> ty *)
fun mapTyTD f tyT = case f tyT of
                      TyTerm(t,ts)       => TyTerm (t, map (mapTyTD f) ts)
...


(* Folds the specified argument using the specified function f.
 * foldTy: (ty * 'a) -> 'a  -> (ty * 'a) -> 'a *)
fun foldTy f (p as TyTerm ( _ ,ts    ), z) = f(p, foldl (foldTy f) z ts)
...

fun toString  TyBool            = "bool"
...

fun pp x = toString x

fun envToString (env: context) = ...

(* Propagates the bindings from the typing context into the argument type.
 * applySubst: context -> ty -> ty *)
fun applySubst env tyIn = ...
fun nextVar() = let val i = !nextSym
                in  nextSym := i + 1;
                    TyVar i
                end

(* Creates a new context with fresh bindings and propagates it.
 * alphaRename: ty -> ty *)
fun alphaRename ty1
      = applySubst
        (foldTy (fn (TyVar x, e) => solve [(TyVar x, nextVar())] e
                  | (_,       e) => e
                )
                (ty1,initialEnv)
        )
        ty1
```

```
(* Records implicitly declared match vars:
 *   for each new <t>_x records a mapping <t>_x -> t[ 'a ].
 * updateEnv: EXPR -> context -> context *)
fun updateEnv expr env
    = foldExpr
        (fn (_, e) => e)
        (foldTerm (fn (imatch_var(sym, id1, _), e) =>
          if S.inDomain(e, #2(symbol (sym^id1)))
          then e
          else S.insert(e, #2(symbol (sym^id1)), nextVar())
                   | (_,e) => e
                 )
        )
        (expr, env)
```

6.4 Unification

Type analysis often requires a check of whether two types are equal and, if not, whether they are unifiable. The following functions provide the implementation of unification. We use SML's equality operator $\cdot = \cdot$ to test whether two types are equal.

```
exception UNIFY;
(* solve: (ty * ty) list * env -> env *)
fun solve [] e = e
  | solve ((TyVar i,     ty2        )::xs) e = elim [(i,ty2)] xs e
  | solve ((ty1,         TyVar i    )::xs) e = elim [(i,ty1)] xs e
  | solve ((TySum (a,b), TySum (c,d))::xs) e = solve ((a,c)::(b,d)::xs) e
...
```

```
  | solve ((a as TyTerm (x,xs),
            b as TyTerm (y,ys))::rest) e = ...
...
  | solve ((TyRule _,        TyTerm _        )::xs) e = raise UNIFY
  | solve ((TyTerm(_,[ty1]), TyBool          )::xs) e
    = solve ((ty1,TyBool  )::xs) e
...
  | solve (( ty1,            ty2             )::xs) e
    = if ty1 = ty2 then solve xs e else raise UNIFY

(* elim: (int * ty) list -> (ty * ty) list -> env -> env *)
and elim []          tts e = solve tts e
  | elim ((x,t)::xs) tts e = let ...


(* Unifies a type with a list of expected types.
 * The first successful unification returns an updated context.
 * unify: (ty * ty list * context * expr) -> context *)
fun unify (givenTy, [expectedTy], env, expr)
      = ((solve [(givenTy,expectedTy)] env) handle
          UNIFY => raiseOperandError expectedTy givenTy expr
                     "operator and operand don't agree")
  | unify (givenTy, expectedTypes, env, expr)
      = let
            fun polyUnify  [] = raisePolyOpError (givenTy, expr)
              | polyUnify (expTy::rest)
                = (solve [(givenTy,expTy)] env) handle
                   UNIFY => polyUnify rest
        in
```

```
            polyUnify expectedTypes
        end
```

6.5 Subtyping

The following functions implement the algorithms of subtyping, calculation of joins and meets, reach-ability of a strategy within a composition, and intersection of typing contexts. The implementations directly correspond to their formal definitions discussed in Chapters 4 and 5.

```
fun subtype _ (TyVar _) = true
  | subtype (TyTerm (x,xs)) (TyTerm (y,ys)) = x = y andalso subtypeL xs ys
  | subtype (ab as TyRule (a,b)) (cd as TyRule (c,d))
      = let
            val _ = finest("subtype. input: " ^ pp ab ^ " | " ^ pp cd) 1
            val out = subtype c a andalso subtype b d
            val _ = finest("subtype.output: " ^ Bool.toString out) ~1
        in
            out
        end
  | subtype a b = if a = b then true else false
(* subtypeL: ty list -> ty list -> bool *)
and subtypeL [] [] = true
  | subtypeL [] _  = false
  | subtypeL _  [] = false
  | subtypeL (x::xs) (y::ys) = subtype x y andalso subtypeL xs ys

(* Converts a given term to a more generic term.
 * generify: ty -> ty *)
fun generify (TyBool)   = TyBool
```

```
...
  | generify (TyVar x)  = TyVar x
  | generify (TyTerm(x,xs))
      = let
            val xs2 = generifyL xs
        in
            if xs = xs2 then nextVar()
            else TyTerm(x,xs2)
        end
  | generify t = raise TypeError("GRAMMAR.generify.match: " ^ pp t)
(* generifyL: ty list -> ty list *)
and generifyL [] = []
  | generifyL (x::xs)
      = let
            val x2 = generify x
        in
            if x = x2 then x::(generifyL xs)
            else x2::xs
        end

exception NoMeetFound
(* Computes the lowest upper bound of two types.
 * join: ty -> ty -> ty *)
fun join (a as TyRule(a1,a2)) (b as TyRule(b1,b2))
    = let
          val _ = finer("join.rules.input: " ^ pp a ^ " | " ^ pp b) 1
          val (nmf, m1) = (false, meet a1 b1) handle
                           NoMeetFound => (true, nextVar())
          val _ = finer("join.rules.args-meet-at: " ^ pp m1) 0
          val j1 = join a2 b2
          val _ = finer("join.rules.outputs-join-at: " ^ pp j1) 0
          val tyF = if nmf then m1 else TyRule(m1, j1)
          val _ = finer("join.rules.output: " ^ pp tyF) ~1
      in
          tyF
      end
  | join a b
    = if subtype a b then b
      else if subtype b a then a
      else let
               val a2 = generify a
               val _ = finer("join.fst: " ^ pp a2) 2
           in
               if pp a = pp a2
               then let
                        val b2 = generify b
                        val _ = finer("join.snd: " ^ pp b2) 2
                    in
                        join b2 a
                    end
               else join b a2
           end
and
    meet (a as TyRule(a1,a2)) (b as TyRule(b1,b2))
    = let
          val _ = finer("meet.rules.input: " ^ pp a ^ " | " ^ pp b) 1
          val j1 = join a1 b1
          val _ = finer("meet.rules.args-join-at: " ^ pp j1) 0
          val m1 = meet a2 b2
          val _ = finer("meet.rules.outputs-meet-at: " ^ pp m1) 0
          val tyF = TyRule(j1, m1)
          val _ = finer("join.rules.output: " ^ pp tyF) ~1
      in
          tyF
      end
  | meet a b
    = if subtype a b then a
      else if subtype b a then b
      else raise NoMeetFound

(* Checks that the 2nd argument is reachable in conditional composition:
 * isReachable: ty -> ty -> bool *)
fun isReachable (a as (TyRule (ty1,_))) (b as (TyRule (ty3,_)))
    = let
          val _ = fine("isReachable input: " ^ pp a ^ " | " ^ pp b) 1
          val result = not (subtype ty3 ty1)
          val _ = fine("isReachable output: " ^ Bool.toString result) ~1
      in
          result
      end
  | isReachable a (TySum (c,d)) = (fine("isReachable.TyRule.TySum") 0;
                                   isReachable a c andalso isReachable a d)
  | isReachable (TySum (a,b)) c = (fine("isReachable.TySum.TyRule") 0;
                                   isReachable a c andalso isReachable b c)
  | isReachable TyInf _ = (fine("isReachable.unsupported any") 0; true)
  | isReachable _ TyInf = (fine("isReachable.any unsupported") 0; true)
  | isReachable _ _ = (fine("isReachable.any") 0; false)
```

```
(* Computes intersection of two contexts; if a key that is present in both
 * contexts is mapped to two distinct types, then the key is mapped to a
 * join of the two types (lowest upper bound) in the result.
 * intersect: (context * context) -> context *)
fun intersect (env1,env2) = ...
```

6.6 Analysis of Composition

The following functions implement the analysis of composition.

```
fun checkStar (TyRule(a,b as TyTerm _), r2 as TyRule(c,d as TyTerm _)) e p
      = let
            val e2 = unify (b, [c], e, p)
                     (* on success, b and c are type-unifiable *)
            val ad = TyRule(applySubst e2 a, applySubst e2 d)
        in
            (ad, e2)
        end
  | checkStar (TyRule(a,b), TyRule(c,d)) e p
      = let
            val e2 = unify (a, [c], e, p)
            val rhs = case b of
                TyRule _  => TyList [b,d]
              | TyList bs => TyList (bs@[d])
              | TyInf     => TyList [b,d]
              | arg       => raise
              TypeError("COMPOSITION.checkStar.rule-rule.match: " ^ pp arg)
            val (a2, rhs2) = (applySubst e2 a, mapTy (applySubst e2) rhs)
            val combTy = TyRule (a2, rhs2)
        in
            (combTy, e2)
        end
...
fun checkSeq (r1 as TyRule(a,b as TyTerm _),
              r2 as TyRule(c,d as TyTerm _)) e p
      = let
            val _ = debug("checkSeq(first-order): "^pp r1^" and "^pp r2) 1
            val (unifiable, e2) = (true, unify (c, [b], e, p)) handle
                                   TypeError _ => (false, e)
            val ad = TyRule(applySubst e2 a, applySubst e2 d)
            val (tyOut, envOut)
               = if unifiable
                 then (TySum(ad, TySum(alphaRename r1, r2)), e2)
                 else (TySum(r1, r2), e)
            val _ = debug("checkSeq result: " ^ pp tyOut) ~1
        in
            (tyOut, envOut)
        end
...
(* Composes strategies x and y using combinator F.
 * createComp: string -> ty -> ty -> context -> expr -> ty *)
fun createComp F x y e p = case F of
                               "<+>" => (TySum (x,y), e)
                             | "<+"  => (TySum (x,y), e)
                             | "+>"  => (TySum (y,x), e)
                             | "<;"  => checkSeq (x,y) e p
                             | ";>"  => checkSeq (y,x) e p
                             | "<*"  => checkStar(x,y) e p
```

```
                              | _     => checkStar(y,x) e p


(* Folds a list of dynamic rules into a single strategy.
 * foldRules: string -> expr -> (ty * context) -> (ty * context) *)
fun foldRules F _ (TyRule(a, b as TyRule _  ), e) = (TyRule (a, b), e)
  | foldRules F p (TyRule(a, TyList (x::xs) ), e)
    = let
          val (ty1,e1)
            = foldl (fn (z,acc) => createComp F (#1 acc) z (#2 acc) p)
                          (x,e) xs
      in
          (TyRule (applySubst e1 a, ty1), e1)
      end
  | foldRules F p (TySum (a,b), e)
    = let
          val (tyA, _) = foldRules F p (a,e)
          val (tyB, _) = foldRules F p (b,e)
      in
          (TySum(tyA, tyB), e)
      end
  | foldRules _ _ (TyInf, e) = (TyInf, e)
  | foldRules _ _ (ty1,_)
    = raise TypeError("COMPOSITION.foldRules.match: " ^ pp ty1)
```

6.7 Analysis of Application

Function reduce implements the analysis of application.

```
(* Computes the type of applying a strategy to a term.
 * reduce: (ty * ty * ctx) -> EXPR -> ty *)
```

```
fun reduce(TyInf, _, e) _ = (TyInf, e)
  | reduce(r as TyRule (_, TyList _), t, _) p
    = raiseAppError r t p "applying a higher-order rule without fold/foldS"
  | reduce(s as TyVar _, t, e) p
    (* applying a strategy of unknown type; perform type-inferencing *)
    = let
          val _ = debug("reduce.tyvar.input: " ^ pp s ^ " | " ^ pp t) 1
          val s2 = TyRule(nextVar(), nextVar())
          val (tyF, e2) = reduce(s2, t, e) p
          val eF = if !precision = 1
                   then e2
                   else unify(s, [applySubst e2 s2], e2, p)
          val _ = debug("reduce.tyvar.inferred: " ^ pp (applySubst eF s)) 0
          val _ = debug("reduce.tyvar.output: result is " ^ pp tyF) ~1
      in
          (tyF, eF)
      end
  | reduce(r as TyRule (lhs,rhs), t, e) p
    = let
          val _ = debug("reduce.rule.input: " ^ pp r ^ " | " ^ pp t) 1
          val e2 = unify (t, [lhs], e, p)
          val rhs2 = applySubst e2 rhs
          val (tyF, eF) = (TySum (rhs2,t), e2)
          val _ = debug("reduce.rule.output: " ^ pp tyF) ~1
      in
          (tyF, eF)
      end
  | reduce(s as TySum (a,b), t, e) p
    = let (* skip rules with incompatible types *)
```

```
          val (illTyped1, (rhs1, e1)) = (false, reduce (a,t,e) p) handle
                                         TypeError _ => (true, (t, e))
          val (illTyped2, (rhs2, e2)) = (false, reduce (b,t,e) p) handle
                                         TypeError _ => (true, (t, e))
          val e12 = intersect (e1,e2)
      in
          if illTyped1 andalso illTyped2 then
            raiseAppError s t p
              "unexpected arguments in strategy application"
          else if illTyped1 then (rhs2, e2)
          else if illTyped2 then (rhs1, e1)
          else (TySum (rhs1, rhs2), e12)
      end
...
```

6.8 Incremental Precision and Verbosity

Application of rewrite rule r to input term t can produce either a new term t' or failure. In *TL* , failure is handled from the identity-based perspective such that the input term is returned unchanged. Thus, static analysis of an application results two new terms – t' in case of success and t in case of failure. This presents significant challenges in the implementation, execution and validation of type analysis, because a linear increase in the number of applications leads to *exponential* increase in the number of possible outcomes.

To cope with the complexities, the type analysis is divided into increasing levels of precision such that lower analysis levels report obvious errors with high degree of confidence and higher analysis levels report finer errors with a correspondingly lower degree of confidence (e.g. a reported error might not actually be a true error).

The precision levels are defined in terms of constructs of the language targeted by the type analysis:

Level 0 No analysis is performed

Level 1 Rewrite rules only

Level 2 Level 1 and combinators

Level 3 Level 2 and fold operators

Level 4 Level 3 and one-layer iterators (`mapL`, `mapR` and `mapB`)

Level 5 Level 4 and full term traversals (e.g. `TDL`, `lcond_tdl`)

Higher precision levels include analysis of all constructs at the lower levels. At a given level, all constructs of upper levels are ignored.

The control over desired precision is given to the *TL* programmer. Thus, for a given program, one can set the type-checking precision and observe the results of type analysis at varying levels of precision.

Similar to the precision levels, the type checker provides logging facilities of incremental verbosity. During analysis, a *TL* programmer can choose the amount of feedback that is printed to the standard console. This allows one to observe the details of type-checking and to obtain greater feedback in case a type error is reported.

The following functions define the logging facilities.

```
val precision = ref 3;


(* Verbosity level:
     0: OFF:     no logging
     1: ERROR:   log only error messages
     2: WARNING: level 1 and warnings
     3: INFO:    level 2 and info messages
     4: DEBUG:   level 3 and debug messages
     5: FINE:    level 4 and fine messages
     6: FINER:   level 5 and finer messages
     7: FINEST:  level 6 and finest messages
     8: ALL:     log all messages *)
val verbosity = ref 3
val indent = ref ""
fun incIndent() = indent := !indent ^ " "
fun decIndent() = indent := implode(tl(explode(!indent)))
fun decode flag msg
    = case flag of
        0 => println(!indent ^ msg)
      | 1 => (incIndent(); println(!indent ^ msg))
      | 2 => (incIndent(); println(!indent ^ msg); decIndent())
      | _ => (println(!indent ^ msg); decIndent())
fun error   msg flag = if !verbosity < 1 then () else decode flag msg
fun warning msg flag = if !verbosity < 2 then () else decode flag msg
fun info    msg flag = if !verbosity < 3 then () else decode flag msg
fun debug   msg flag = if !verbosity < 4 then () else decode flag msg
fun fine    msg flag = if !verbosity < 5 then () else decode flag msg
fun finer   msg flag = if !verbosity < 6 then () else decode flag msg
fun finest  msg flag = if !verbosity < 7 then () else decode flag msg
fun all     msg flag = if !verbosity < 8 then () else decode flag msg

type configuration = {precision: int, verbosity: int}

fun init prec verb
    = ((* set to false to skip assertion-checking/regression-testing *)
       inTestingMode := false;
       (* precision of type-checking from 0 to 5 in the increasing order *)
       precision := prec;
       (* verbosity of output from 0 to 6 in the increasing order *)
       verbosity := verb)
```

6.9 Type-checking

Having surveyed the key auxiliary components of type analysis, we are now ready to discuss the functions that drive the analysis.

The type-checking function `typeOf` is in direct correspondence with typing rules that were presented in Chapter 4 and 5. In particular, for each kind of constructor defined in Section 6.1, the function computes the types of immediate sub-expressions and builds an output type if everything fits or raises a type error otherwise.

```
fun typeOf(Bool   _, e) = (  TyBool, e)
  | typeOf(Int    _, e) = (   TyInt, e)
  | typeOf(Real   _, e) = (  TyReal, e)
  | typeOf(String _, e) = (TyString, e)
  | typeOf(Identifier (id, ninfo), e)
    = (case S.find(e, #2(symbol id)) of
        SOME (TyVar i) => (TyVar i, e)
      | SOME type1 => (fine("typeOf.id: " ^ id ^ " | " ^ toString type1) 2;
                      (alphaRename type1, e))
      | NONE       => raise TypeError(ninfoToString ninfo ^
                            "Type error: undefined variable: " ^ id) )
  | typeOf(p as Concat  (expr1, expr2, _), e)
      = let
            val (ty1, e1) = typeOf(expr1, e )
            val (ty2, e2) = typeOf(expr2, e1)
```

```
          val e3 = unify (TySum (ty1,ty2),
                          [TySum (TyString,TyString)], e2, p)
      in
          (replaceLeaf [TyString]
           (pickATree (applySubst e3 ty1, applySubst e3 ty2)), e3)
      end
| typeOf(p as Plus    (expr1, expr2, _), e)
    = let
          val (ty1, e1) = typeOf(expr1, e )
          val (ty2, e2) = typeOf(expr2, e1)
          val e3 = unify (TySum (ty1,ty2),
                          [TySum (TyInt,TyInt),
                           TySum (TyReal,TyReal)], e2, p)
      in
          (replaceLeaf [TyInt,TyReal]
           (pickATree (applySubst e3 ty1, applySubst e3 ty2)), e3)
      end
| typeOf(p as Term (aTree,ninfo), e)
  = let
      fun getTermType (itree (inode(name, _), [])) = TyTerm (name, [])
        | getTermType (itree (imatch_var (sym,id1, _), []))
            = (case S.find(e, #2(symbol (sym^id1))) of
                 SOME (TyTerm x) => TyTerm x
               | SOME primTy => TyTerm(sym, [primTy])
               | NONE        =>
                 raise TypeError(ninfoToString ninfo ^
                    "Type error: unbound/free pattern variable " ^
                    "<" ^ sym ^ ">" ^ id1 ^ crlf ^
                    "  in expression: " ^ exprToString p))
        | getTermType (itree (inode(name,_), xs))
          = TyTerm (name, map getTermType xs)
        | getTermType (itree (imatch_var _,_))
          = raise TypeError "Impossible: match vars cannot have subterms"
      val ty1 = getTermType aTree
    in
      (ty1, e)
    end
| typeOf(p as Rule      (left,right,_), e)
  = let
      val _ = fine("typeOf.rule.raw-input: " ^ exprToString p) 2
      val e0 = updateEnv left e (* bind schema vars on the left*)
      val (ty1, e1) = typeOf(left,  e0) (* infer types *)
      val (ty2, e2) = typeOf(right, e1)
      val r1 = TyRule (ty1, ty2)
      val _ = debug("typeOf.rule.input: " ^ toString r1) 1
      val tyOut = applySubst e2 r1 (* propagate type inferences *)
      val _ = debug("typeOf.rule.output: " ^ toString tyOut) ~1
    in
          (tyOut, e2)
    end
| typeOf(Id _,   e) = let val a = nextVar()
                      in  (TyRule(a, a), e)
                      end
| typeOf(p as LChoice (a, b, _), e)
    = let
          val (ty1, _) = typeOf(a, e)
          val (ty2, _) = typeOf(b, e)
          val _ = if isReachable ty1 ty2
```

```
                  then ()
                  else raiseCompositionError ty1 ty2 p
                       "right operand is subsumed by the left operand"
      in
          if !precision < 2 then (TyInf, e)
          else (TySum (ty1,ty2), e)
      end
| typeOf(p as LStar   (left,right,_), e)
    = let
          val (ty1, e1) = typeOf(left, e )
          val (ty2, e2) = typeOf(right,e1)
      in
          if !precision < 2 then (TyInf, e)
          else checkStar (ty1,ty2) e2 p
      end
| typeOf(p as LSeq    (left,right,_), e)
    = let
          val (ty1, e1) = typeOf(left, e )
          val (ty2, e2) = typeOf(right,e1)
      in
          if !precision < 2 then (TyInf, e)
          else checkSeq (ty1,ty2) e2 p
      end
| typeOf(p as FoldChoice   (expr,_), e)
    = let
          val (ty1, e1) = typeOf(expr, e)
      in
          if !precision < 3 then (TyInf, e)
          else foldRules "<+>" p (ty1, e1)
```

```
        end
  | typeOf(p as MapL (expr,_), e)
      = let
            val (ty1, e1) = typeOf(expr, e)
        in
            if !precision < 4 then (TyInf, e)
            else (TyMap ty1, e1)
        end
  | typeOf(p as Iterator  (Identifier (id,_), [s], _), e)
      = let
            val (ty1, e1) = typeOf(s, e)
        in
            if !precision < 5 then (TyInf, e)
            else (TyIter (id,ty1), e1)
        end
...
```

Whereas function `typeOf` computes types of expressions, the following function processes declarations and updates a given context with a new mapping for a declared variable.

```
(* Checks recursive and non-recursive declarations.
 * tyCheck: context -> EXPR -> context *)
fun tyCheck e (Prog (declList,_) )
    = foldl (fn (x,ctx) => tyCheck ctx x) e declList
  | tyCheck e (p as NonRecursive (Identifier (id,_), expr, ninfo))
    = let
            val _ = debug("tyCheck: analyzing body of " ^ id) 1
            val (ty1, _) = typeOf(expr, e) handle
              TypeError msg => (error("***" ^ crlf ^ msg ^ crlf) 0;
                                incErrors();
                                (TyError, e))
            val _ = debug("tyCheck: computed type is " ^ toString ty1) 0
      in
           enter(e, id, ty1)
      end
  | tyCheck e (Recursive (Identifier (id,_), args, expr, _))
    = enter(e, id, TyInf)
  | tyCheck _ expr
    = raise TypeError("TYPECHECK.tyCheck.match: " ^ exprToString expr)
```

Finally, function typeCheck is the main driver function that analyzes a program starting from an empty context and reports errors, if any.

```
(* Given
 *   - smlDecls: declarations of types of SML functions
 *   - program: TL program as an abstract syntax tree
 *   - grammar: extended-BNF grammar of a term language
 *   - configuration parameters defining type-checker's
 *       precision and verbosity levels
 * computes types of rewrite rules and recursive definitions and determines
 * if the TL program is well-typed.
 * typeCheck: EXPR -> EXPR -> grammar ->
              {precision: int, verbosity: int} -> EXPR *)
fun typeCheck smlDecls
              program
              (G.GRAMMAR {precassoc_rules, production_list})
              {precision, verbosity}
  = if precision = 0 then program else
    let
    val _ = init precision verbosity
    val _ = info "[starting type analysis]" 0
    val program1 = ckDuplicates program
    val program2 = removeFwdRefs program1
    val _ = inputTree := program2
    val _ = tgtGrammar := crop production_list (!tgtGrammar)

    val env1 = case smlDecls of
                 SIGNATURE (LIST (expr,_), _) =>
                   getSmlSignatures expr initialEnv
               | _                            => initialEnv
    val env2 = tyCheck env1 program2
                 handle x => (error(exnName x ^ " " ^
                                    exnMessage x) 0; env1)

    val _ = info(envToString env2) 0
in
    if !errors = 0
    then (info "[type analysis completed successfully]" 0; !inputTree)
    else (error("!***" ^ crlf ^
                "    Type analysis failed with " ^
                Int.toString (!errors) ^
                " errors. See error messages above." ^ crlf ^
                "***!" ^ crlf ^
                "[type analysis completed with errors]") 0;
          !inputTree )
end
```

6.10 Performance Experiments

In this section, we present results of some performance measurements of our implementation. In our experiments, we are interested in the time that our type system takes to analyze *TL* programs. Our benchmarks consist of the following programs written in *TL* :

Regression Test Suite consists of a collection of unit tests used to check correctness of the type analysis. Each unit test is represented by a labeled strategy s (e.g. $s : id[\![x]\!] \rightarrow id[\![y]\!]$) and a corresponding type assertion in the form of a labeled string *assert_s* (e.g. $assert\_s$: "$id[x] \rightarrow id[y]$"). In testing mode, the type-checker computes the type of a given labeled strategy and compares it with the asserted type for equality modulo type variable indices. An inequality of the two type expressions is reported to a programmer to indicate an error either in the asserted type or the implementation. Thus, the test suite serves an additional purpose of a regression suite for additions and extensions to the existing type-checker. We report on the results of three disjoint test suites testing the type-checker at precision levels 1, 2 and 3.

DSL Compiler is a transformation-based source-to-source compiler of a Domain Specific Language called *Paradigm* into a micro-code program. The purpose of the DSL is to expose micro-code level parallelism via high-level program abstractions. The intended execution platform of DSL programs is *Score*: an embedded hardware realization of the Java Virtual Machine developed and maintained by the Sandia National Laboratories.

Figure 6.4 summarizes measurements of the experiment on the test suites. The execution platform for this experiment was an Intel Core 2 Duo 2.2GHz CPU, 4GB RAM, Windows Vista 32-bit OS, SML/NJ v110.0.7 (Sep 28, 2000). Here, the results are listed for three disjoint test suites that validate the type-checker at three precision

		Analysis Time (sec)			
Precision	SLOC	Wall	CPU	GC	Speed (SLOC/sec)
1	4344	21.289	21.289	0.346	204.049
2	487	0.133	0.133	0.001	3,661.654
3	615	0.432	0.432	0.006	1,423.611
Total	5446	21.854	21.854	0.353	249.199

Figure 6.4: Performance measurements for “Regression Test Suite”

levels. While the test suites are disjoint, the precision levels are accumulating in the sense that at level 3 the type-checker still performs level-1 and level-2 analyses.

The second column (SLOC) lists the number of source lines of code including white-space and comment lines. The next group of columns displays the execution time of the type-checker in seconds from the moment the type analysis is invoked until the moment the control is transferred back to the *TL* 's engine. The execution time is reported in three forms: (1) Wall time, which is the real clock time, (2) CPU time, which is the time the process has had the CPU and (3) GC time, which is the time the process has spent on garbage collection. The final column lists an approximate speed of the analysis in terms of the number of SLOC analyzed per second.

Note that the size of the first test suite is an order of magnitude larger than the other test suites. This explains an order of magnitude decrease in the analysis speed between the first and the other test suites. Next, the second test suite is smaller in size than the third, which explains the faster execution speed on the second test suite. Also, the third test suite performs all three levels of type analysis, which explains why the third test suite while being only 25% larger in size than the second test suite is analyzed at the 40% of the second test suite's analysis speed.

If we add up the lines of code and execution times, then we can observe an approximate overall speed of $5446/21.854 \approx 250$ lines per second. We attribute this speed to the expensiveness of string comparison operations that are extensively used in test suites' comparisons of asserted versus actual types.

Precision	SLOC	Analysis Time (sec) Wall	CPU	GC	Speed (SLOC/sec)
1	9652	2.869	2.869	0.166	3,364.238
2	9652	5.607	5.607	0.463	1,721.419
3	9652	5.536	5.536	0.455	1,743.497
Average	9652	4.671	4.671	0.361	2,066.367

Figure 6.5: Performance measurements for “DSL Compiler”

Figure 6.5 summarizes the analysis time of the DSL compiler. The execution platform for this analysis was Intel Core i7 940 2.93GHz, 8GB RAM, Windows 7 Enterprize 64-bit OS, SML/NJ v110.0.7 (Sep 28, 2000). Here, the same *TL* program has been analyzed at three different precision levels. We can observe that on average the analysis executes at approximately 2000 lines per second.

We attribute an order of magnitude difference in the execution time between the two experiments to the frequency of (expensive) string comparison operations in the first experiment. Since the second experiment more closely approximates the actual usage of the type system, where there are few (if at all) asserted versus actual type comparisons, the results of the second experiment are closer to an actual execution speed. Thus, we conclude that the overall approximate performance of the analysis is on the order of thousands of source lines of code per second.

6.11 Limitations of the Implementation

Due to the complexity of type analysis as discussed in Section 6.8, the implementation does not perform analysis of all syntactic constructs of *TL* . In particular, the current precision of the type-checker that has been tested and validated is Level 3 – up to and including rewrite rules and (higher-order) compositions. We leave the implementation of analysis of term traversals as future work.

The following list provides examples of strategies, whose analysis remains for future work.

One-layer traversals: Partially supported

Applications: Implemented and validated. Example: `(mapL r) t`

Compositions: Partially implemented, not validated. Example:

`(mapL r) <* s` and `(mapL r) <; s`

Full term traversals: Unsupported

Applications Example: `(TDL r) t`

Compositions Example: `(TDL r) <* s` and `(BUL r) <; s`

Traversals with self-modifying higher-order compositions: Unsupported.

Example: `mapL(foldS` $\lhd\!\!+$ `s)`

Custom traversals: Unsupported. Example: `def customTDL s = ...`

Chapter 7

Conclusions and Future work

In this research, we have constructed a type system and presented type analysis of strategic rewriting language *TL* . Without type analysis, programming rewrite strategies in an un-typed language is prone to errors. Errors occur due to complexity of manipulated values, which are parse tree terms exhibiting deep and wide structures. Pattern-matching on complex terms may fail at arbitrary sub-term depths and positions, which makes traditional debugging of logging and comparing input and output terms difficult. In addition to the complexity of terms, failure of a rule to match and rewrite an input term may lead to application attempts of other rewrite rules in a strategic composition's sequence. In the worst case, the number of possible execution paths in rewrite strategies exhibits exponential complexity. Facing such complexities, automated analysis and detection of errors becomes a necessity.

Having surveyed the most common types of errors in rewrite strategies in Chapter 1, we then reviewed type systems and their analysis as a syntax-directed lightweight formal method of proving absence of errors in a program. This was followed by an overview of the syntactic constructs and features of *TL* – a representative transformation language supporting all of the major features of strategic rewriting.

In Chapter 4, we presented the core aspects of type analysis of rewrite strategies.

The distinguishing features of this system are (1) recursive data type representation of a term's type to track the entire term structure for high-precision of type checks, (2) union types to track multiple potential outcomes of a strategic application, and (3) static analysis and type inferencing to minimize changes while integrating the type system into an un-typed language infrastructure.

Having defined the essential features of the analysis, we then turned to the analysis of non-standard features of *TL* such as SML function invocations, primitive operations on terms and higher-order rules and compositions. Here, we have shown how simple extensions of the core system can provide high-precision type checks to the entire language.

As part of this research, the type system has been implemented in SML and integrated into the infrastructure of *TL*. The numerous type-checked examples given throughout the presentation are direct results of the implementation. Besides a several KLOC-size regression test suite, the implementation is currently being used in at least two industrial transformation projects, one of which has reported a previously uncaught error detected by the type system.

In closing, this research produced a practical high-precision type analysis framework for detection of errors in rewrite strategies. Programmers of rewrite strategies, assisted with automated type analysis, no longer need to solely rely on testing to determine whether a strategy is valid and can succeed in rewriting of inputs. Similar to the success of type systems in making mainstream programming easier, it is the goal of this research to have made strategic rewriting easier.

7.1 Future Work

Due to the size and complexity of analyzing all features of a strategic programming language – rewrite rules, rule compositions and term traversals, the following aspects of the analysis remain as part of future work:

1. The analysis needs to be extended to include analysis of applicability of full-term traversals. Current analysis capabilities can detect whether a strategy can succeed in application to direct sub-terms of a term: e.g. `mapL s t`. This remains to be extended to full-term traversals: e.g. `TDL s t`. Such analysis would flag traversals that have no chance of succeeding. In turn, elimination of such traversals could lead to performance improvements in the execution time of a transformation program.

2. The analysis needs to be extended to include full-scale analysis of traversal compositions. In particular, future work needs to lift success or failure of application of a single rewrite rule to the success or failure of application of a traversal of a term with a rewrite rule. This kind of analysis would enable determination of whether one-layer traversal compositions (e.g. `mapL r` <* `s`) or full-term traversal compositions (e.g. `TDL r` <* `s`) can succeed. Elimination of such traversals can also lead to performance improvements.

3. *TL* provides a heterogeneous operator `foldS` that allows a programmer to fold dynamically generated strategies into the strategy that is used by a traversal: e.g. `mapL(foldS` <+ `s)`. In essence, this is an example of a traversal with a self-modifying higher-order strategy, where a newly generated strategy is folded to the left of the existing strategy: i.e. `mapL(foldS` <+ `(s`$_n$ <+ `...` <+ `s`$_1$ <+ `s))`. The details of this analysis need to include detection of whether the original strategy `s` can succeed within a traversal and whether the dynamically generated strategy can succeed within the traversal.

4. *TL* allows a programmer to define custom traversals using recursive equations of the form `def customTraversal s = ...`. Analysis of traversals needs to be extended to include custom traversal definitions. In particular, the analysis

could determine whether a given custom traversal definition can succeed when applied to a term with a given strategy or whether a custom traversal can succeed at all.

5. The full proof of type soundness of the analysis of all *TL* features.

6. Analysis of traversals can significantly increase complexity of the analysis. Future work needs to investigate a possibility of abstractions of types. In particular, if a given non-terminal symbol of a term language grammar can derive several combinations of sub-term symbols, it could be possible to investigate applicability of a strategy to the symbol itself, instead of the multiple variants of sub-term symbols. This includes (a) pruning a type's tree structure toward its root and (b) combining multiple constructor alternatives derivable from the same parent symbol into one parent symbol (e.g. $[Int(...), Float(...)]$ into just $[Number(TyVar(i))]$).

7. In addition to the abstraction of types outlined above, future work includes an overall investigation of performance improvements. In particular, related work has identified the use of tree automata in the optimization of type-checking of regular expression types. Similar performance improvements can potentially be obtained in the type-checking based on constructor-based types used in this research.

Appendix A

Boolean expressions

Below is the implementation of an interpreter of boolean expressions in *TL*.

Target language

```
USE_LR_PARSER

t

<t>  ::= <v>

       | "if" <t> "then" <t> "else" <t> .

<v>  ::= "true"

       | "false"        .
```

Program

```
main: FIX oneStep


oneStep:

    <t>_in -> <t>_out

    if { s :=

            t[:] if true then <t>_2 else <t>_3 [:] -> <t>_2

            <+

            t[:] if false then <t>_2 else <t>_3 [:] -> <t>_3

            <+

            t[:] if <t>_1a then <t>_2 else <t>_3 [:] ->

            t[:] if <t>_1b then <t>_2 else <t>_3 [:]
            if {
                <t>_1b = s <t>_1a
            }
    andalso
        <t>_out = s <t>_in
}
```

Test

▷(*Input*) `if if if true then false else false then true else true then true else true`

◁(*Output*) `true`

Appendix B

Arithmetic expressions

Below is the implementation of an interpreter of arithmetic expressions in *TL*.

Target language

```
USE_LR_PARSER

t

<t> ::= <v>

      | "if" <t> "then" <t> "else" <t>

      | "succ" <t>

      | "pred" <t>

      | "iszero" <t>.

<v> ::= "true"

      | "false"

      | <nv>        .

<nv>::= "0"         .
```

Program

```
main: FIX oneStep


oneStep:

    <t>_in -> <t>_out

    if { s :=

            // reduction of booleans
            t[:] if true then <t>_2 else <t>_3 [:] -> <t>_2
            <+
            t[:] if false then <t>_2 else <t>_3 [:] -> <t>_3
            <+
            t[:] if <t>_1 then <t>_2 else <t>_3 [:] ->
            t[:] if <t>_11 then <t>_2 else <t>_3 [:]
                if { <t>_11 = s <t>_1 }

            // reduction of arithmetic exprs
            <+
            t[:] pred 0 [:] -> t[:] 0 [:]
            <+
            t[:] pred succ <t>_1 [:] -> <t>_1
            <+
            t[:] iszero 0 [:] -> t[:] true [:]
            <+
            t[:] iszero succ <t>_1 [:] -> t[:] false [:]
            <+
            t[:] succ <t>_1 [:] -> t[:] succ <t>_11 [:]
                if { <t>_11 = s <t>_1 }
            <+
            t[:] pred <t>_1 [:] -> t[:] pred <t>_11 [:]
                if { <t>_11 = s <t>_1 }
            <+
            t[:] iszero <t>_1 [:] -> t[:] iszero <t>_11 [:]
                if { <t>_11 = s <t>_1 }
        andalso
            <t>_out = s <t>_in
    }
```

Test

```
▷  if iszero pred 0 then 0 else succ 0
◁  0
```

Appendix C

Lambda calculus

Below is the implementation of an interpreter of lambda calculus expressions in *TL*.

Target language

```
USE_LR_PARSER
t
<t> ::= <id>
      | <v>
      | "(" <t> <t> ")" .
<v> ::= "(" "lam" <id> "." <t> ")" .
<id>::= lexId .
```

Program

```
main: FIX oneStep


oneStep:
    <t>_in -> <t>_out
    if { s :=
                t[:] ( ( lam <id>_x . <t>_1 ) <v>_1 ) [:] ->
                TDL (t[:] <id>_x [:] -> t[:] <v>_1 [:] ) <t>_1
            <+
                t[:] ((<t>_1a <t>_1b) <t>_2) [:] -> t[:] (<t>_11 <t>_2) [:]
                    if {
```

```
                    <t>_11 = s t[:] (<t>_1a <t>_1b) [:]
                }
        <+
            t[:] (<v>_1 (<t>_2a <t>_2b)) [:] -> t[:] (<v>_1 <t>_22) [:]
                if {
                    <t>_22 = s t[:] (<t>_2a <t>_2b) [:]
                }
    andalso
        <t>_out = s <t>_in
}
```

Test

1. Test Church boolean true

```
▷  (((lam t.  (lam f.  t)) (lam a.  a)) (lam b.  b))
◁  ( lam a .  a )
```

2. Test Church boolean false

```
▷  (((lam t.  (lam f.  f)) (lam a.  a)) (lam b.  b))
◁  ( lam b .  b )
```

3. Test (and true true)

```
▷  (((lam b.  (lam c.  ((b c) (lam t.  (lam f.  f)) )))
       (lam t1.  (lam f1.  t1))) (lam t2.  (lam f2.  t2)))
◁  ( lam t2 .  ( lam f2 .  t2 ) )
```

4. Test (and false true)

```
▷  (((lam b.  (lam c.  ((b c) (lam t.  (lam f.  f)) )))
       (lam t1.  (lam f1.  f1))) (lam t2.  (lam f2.  t2)))
◁  ( lam t .  ( lam f .  f ) )
```

5. Test (fst (pair v w))

```
▷  ((lam p.  (p (lam ta.  (lam fa.  ta))))
       (((lam f.  (lam s.  (lam b.  ((b f) s)) ))
          (lam v.  v)) (lam w.  w)) )
◁  ( lam v .  v )
```

Appendix D

Typed arithmetic expressions

Below is the implementation of a type-checker of arithmetic expressions in *TL*.

Target language

```
USE_LR_PARSER

t

<t> ::= <v>

      | "if" <t> "then" <t> "else" <t>

      | "succ" <t>

      | "pred" <t>

      | "iszero" <t>

      | <ty> .

<v> ::= "true"

      | "false"

      | <nv> .

<nv>::= "0" .

<ty>::= "BOOL"

      | "NAT"

      | "ABORT" .
```

Program

```
step:

    <t>_in -> <t>_out
    if { s :=
            t[:] true [:] -> t[:] BOOL [:]
            <+
            t[:] false [:] -> t[:] BOOL [:]
            <+
            t[:] if <t>_1 then <t>_2 else <t>_3 [:] -> <t>_then
                if {t[:] BOOL [:] = s <t>_1 andalso
                    <t>_then = s <t>_2 andalso
                    <t>_else = s <t>_3 andalso
                    <t>_then = <t>_else
                }
            <+
            t[:] 0 [:] -> t[:] NAT [:]
            <+
            t[:] succ <t>_1 [:] -> t[:] NAT [:] if { t[:] NAT [:] = s <t>_1 }
            <+
            t[:] pred <t>_1 [:] -> t[:] NAT [:] if { t[:] NAT [:] = s <t>_1 }
            <+
            t[:] iszero <t>_1 [:] -> t[:] BOOL [:] if { t[:] NAT [:] = s <t>_1 }
        andalso
            <t>_out = s <t>_in
    }

main:
    <t>_in -> <t>_out
    if { <t>_1 = FIX step <t>_in andalso
         (((<t>_1 = t[:] BOOL [:] orelse <t>_1 = t[:] NAT [:])
          andalso <t>_out = <t>_1)
         orelse <t>_out = t[:] ABORT [:])
    }
```

Test

```
▷ if iszero pred 0 then 0 else succ 0
◁ NAT
```

```
▷  iszero true

◁  ABORT
```

Appendix E

Simply typed lambda-calculus

Below is the implementation of a type-checker of lambda calculus expressions in *TL*.

Target language

```
USE_LR_PARSER

t

<t> ::= <id>

      | <v>

      | "(" <t> <t> ")"

      | <ty> .

<v> ::= "(" "lam" <id> ":" <ty> "." <t> ")" .

<id>::= lexId .


<ty>::= <tybase>

      | <tybase> "->" <ty>

      | "(" <ty> ")" "->" <ty> .

<tybase>

    ::= lexTy

      | "ABORT" .
```

Program

```
step:

  <t>_in -> <t>_out
  if {
    s :=
      t[:] ( lam <id>_x : <ty>_x . <t>_1 ) [:] -> <t>_3
        if {
          <t>_2 = TDL (<id>_x -> <ty>_x) <t>_1 andalso
          t[:] <ty>_y [:] = s <t>_2 andalso
          ((ty[:] <tybase>_x [:] = <ty>_x andalso
           <t>_3 = t[:] <tybase>_x -> <ty>_y [:]) orelse
           <t>_3 = t[:] (<ty>_x) -> <ty>_y [:])
        }
      <+
      t[:] (<t>_1 <t>_2) [:] -> t[:] <ty>_2 [:]
        if {
          <t>_11 = s <t>_1 andalso
          <t>_22 = s <t>_2 andalso
          ((t[:] (<ty>_1) -> <ty>_2 [:] = <t>_11 andalso
           t[:] <ty>_1 [:] = <t>_22) orelse
           (t[:] <tybase>_1 -> <ty>_2 [:] = <t>_11 andalso
            t[:] <tybase>_1 [:] = <t>_22))
        }
    andalso
      <t>_out = s <t>_in
  }


main:
  <t>_in -> <t>_out
    if { <t>_1 = FIX step <t>_in andalso
         ((<t>_1 = <t>_in andalso <t>_out = t[:] ABORT [:]) orelse
          <t>_out = <t>_1)
    }
```

Test

1. The Church boolean true

```
▷  (lam t:  A. (lam f:  B. t))
◁  A -> B -> A
```

2. Test the Church boolean true

```
▷  (((lam t:  T -> T. (lam f:  F -> F. t)) (lam a:  T. a)) (lam b:  F. b))
◁  T -> T
```

3. Ill-typed test of the Church boolean true

```
▷  (((lam t:  T. (lam f:  F. t)) (lam a:  T. a)) (lam b:  F. b))
◁  ABORT
```

4. Higher-order function

```
▷  (lam f:  A->A. (lam x:  A. (f (f x)) ))
◁  ( A -> A ) -> A -> A
```

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[2] Robert L. Akers, Ira D. Baxter, Michael Mehlich, Brian J. Ellis, and Kenn R. Luecke. Case study: Re-engineering C++ component models via automatic program transformation. *Information and Software Technology*, 49:275–291, 2007.

[3] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, New York, NY, USA, 1998.

[4] Andrew W. Appel, James S. Mattson, and David R. Tarditi. A lexical analyzer generator for Standard ML. Version 1.6.0, October 1994; c1989-2004 [updated 1996 July 22; cited 2010 August 19]. Available from: http://www.smlnj.org/doc/ML-Lex/manual.html.

[5] Malte Appeltauer and Günter Kniesel. Towards Concrete Syntax Patterns for Logic-based Transformation Rules. In *Proceedings of the Eighth International Workshop on Rule-Based Programming (RULE'07)*, Paris, France, 2007.

[6] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[7] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring Support for Class Library Migration. In *Proceedings of OOPSLA 2005*, pages 265–279, San Diego, California, United States, 2005. ACM.

[8] Ira D. Baxter. Parallel Support for Source Code Analysis and Modification. In *Source Code Analysis and Manipulation (SCAM'02)*, 2002.

[9] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS: Program Transformations for Practical Scalable Software Evolution. In *International Conference on Software Engineering (ICSE'04)*, pages 625 – 634. IEEE Press, 2004.

[10] P. Borovansky, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In C. and H. Kirchner, editors, *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications*, September 1998.

[11] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. Program Transformation with Scoped Dynamic Rewrite Rules. *Fundamenta Informaticae*, 69(1–2):123–178, 2006.

[12] Rod M. Bustall and John Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[13] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, Jan 1990.

[14] Horatiu Cirstea and Claude Kirchner. An Introduction to the Rewriting Calculus. Research Report RR-3818, INRIA, DEC 1999.

[15] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

[16] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Principles of Programming Languages (POPL'98)*, San Diego, California, USA, Jan 1998.

[17] Guy Cousineau and Michel Mauny. *The Functional Approach to Programming.* Cambridge University Press, 1998.

[18] Alcino Cunha and Joost Visser. Transformation of structure-shy programs: applied to XPath queries and strategic functions. In *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'07)*, pages 11–20. ACM Press, 2007.

[19] Danny Dig and Ralph Johnson. The Role of Refactorings in API Evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 389 – 398. IEEE, 2005.

[20] Alan Donovan, Adam Kiezun, Matthew S. Tschantz, and Michael D. Ernst. Converting Java Programs to Use Generic Libraries. In *Proceedings of OOPSLA 2004*, pages 15 – 34, Vancouver, BC, Canada, 2004.

[21] Kent R. Dybvig. *The Scheme Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 2 edition, 1996.

[22] Peter H. Eidorff, Fritz Henglein, Christian Mossin, Henning Niss, Morten H.Sorensen, and Mads Tofte. AnnoDomini: From Type Theory to Year 2000 Conversion Tool. In *Proceedings of POPL 1999*, pages 1–14, Austin, Texas, USA, 1999.

[23] J. Nathan Foster, Benjamin C. Pierce, and Alan Schmitt. A Logic Your Typechecker Can Count On: Unordered Tree Types in Practice. In *Proceedings of the ACM SIGPLAN Workshop on Programming Language Technologies for XML (PLAN-X)*, pages 80–90, 2007.

[24] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving The Design of Existing Code*. Addison-Wesley, 1999.

[25] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. Edinburgh LCF. *Springer-Verlag LNCS*, 78, 1979.

[26] James Gosling, Bill Joy, Guy L. Steele Jr., and Gilad Bracha, editors. *The Java Language Specification(3rd edition)*. Sun Microsystems Press, 2006.

[27] Johannes Henkel and Amer Diwan. CatchUp! Capturing and Replaying Refactorings to Support API Evolution. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, St.Louis, Missouri, USA, 2005.

[28] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2nd edition edition, 2000.

[29] Cay S. Horstmann and Gary Cornell, editors. *Core Java 2. Volume I – Fundamentals*. Sun Microsystems Press, 2005.

[30] Haruo Hosoya, Jerome Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27(1):46–90, 2005.

[31] Paul Hudak, S. Peyton Jones, Philip Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language Haskell, version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

[32] S.L. Peyton Jones and A.L.M. Santos. A Transformation-based Optimizer for Haskell. *Science of Computer Programming*, 32(1-3):3–47, Sep 1998.

[33] Markus Kaiser and Ralf Lämmel. An Isabelle/HOL-based Model of Stratego-like Traversal Strategies. In *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'09)*, September 2009.

[34] Deepak Kapur and Hantao Zhang. An Overview of Rewrite Rule Laboratory. In *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications (RTA'89)*, 1989.

[35] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

[36] Richard Kelsey, William Clinger, and Jonathan Rees. Revised report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, September 1998.

[37] Günter Kniesel. A Logic Foundation for Conditional Program Transformations. Technical Report IAI-TR-2006-1, Computer Science Department III, University of Bonn, January 2006. ISSN 0944-8535.

[38] Jan Kort, Ralf Lämmel, and Joost Visser. Functional Transformation Systems. In *Proceedings of the 9th International Workshop on Functional and Logic Programming (WFLP'00)*, Benicassim, Spain, jul 2000.

[39] Vitaly Lagoon and Peter J. Stuckey. A Framework for Analysis of Typed Logic Programs. In Herbert Kuchen and Kazunori Ueda, editors, *Proceedings of 5th International Symposium on Functional and Logic Programming (FLOPS'01)*, volume 2024 of *Lecture Notes in Computer Science*, Tokyo, Japan, March 2001. Springer.

[40] Ralf Lämmel. Typed Generic Traversal with Term Rewriting Strategies. *Journal of Logic and Algebraic Programming*, 54:1–64, 2003.

[41] Ralf Lämmel. Scrap your boilerplate with XPath-like combinators. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 137–142. ACM Press, 2007.

[42] Ralf Lämmel, Simon Thompson, and Markus Kaiser. Programming errors in traversal programs over structured data. *ENTCS*, 238(5):135–153, 2008. Also appears in proceedings of LDTA 2008.

[43] Ralf Lämmel, Eelco Visser, and Joost Visser. The Essence of Strategic Programming. October 2002.

[44] Ralf Lämmel and Joost Visser. Typed Combinators for Generic Traversal. In *Practical Aspects of Declarative Programming (PADL'02)*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, January 2002.

[45] Ralf Lämmel, Joost Visser, and Jan Kort. Dealing with Large Bananas. In Johan Jeuring, editor, *Proceedings of Workshop on Generic Programming (WGP'00), Technical Report, Universiteit Utrecht*, pages 46–59, jul 2000.

[46] Xavier Leroy. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Remy, and Jerome Vouillon. Available from: http://caml.inria.fr.

[47] T. Lindholm and F. Yellin, editors. *The Java Virtual Machine (Second Edition)*. Addison-Wesley, 1999.

[48] Daniel Lohmann, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. On the Design and Development of a Customizable Embedded Operating System. In *Proceedings of the International Workshop on Dependable Embedded Systems, 23rd Symposium on Reliable Distributed Systems (SRDS 2004)*, October 2004.

[49] Azamat Mametjanov and Victor Winter. Type Checking Term-Rewriting Strategies. Technical Report cst-2009-3, University of Nebraska at Omaha, Nov 2009.

[50] Azamat Mametjanov, Victor Winter, and Ralf Lämmel. More Precise Typing of Rewrite Strategies. In *Proceedings of the 11th International Workshop on Language Descriptions, Tools, and Applications (LDTA'11)*. ACM, 2011.

[51] J. A. McCoy. An Embedded System for Safe, Secure and Reliable Execution of High Consequence Software. In *HASE 2004: The* 5^{th} *IEEE International Symposium on High Assurance Systems Engineering*, pages 107–114, Albuquerque, New Mexico, United States, 2004. IEEE.

[52] Tom Mens, Gűnter Kniesel, and Olga Runge. Transformation dependency analysis – a comparison of two approaches. In *Proc. of Langages et Modèles à Objets (LMO2006)*, pages 167–182, Mar 22–24 2006. To appear.

[53] Gregor Meyer. On the Use of Types in Logic Programming. Informatik Berichte 199, FernUniversität Hagen, Jun 1996.

[54] Gregor Meyer. Type Checking and Type Inferencing for Logic Programs with Subtypes and Parametric Polymorphism. Informatik Berichte 200, FernUniversität Hagen, Jun 1996.

[55] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[56] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[57] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[58] Pierre-Etienne Moreau and Antoine Reilles. Rules and Strategies in Java. In *Proceedings of the Seventh International Workshop on Reduction Strategies in Rewriting and Programming (WRS'07)*, June 2007.

[59] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[60] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[61] Riccardo Pucella. Notes on Programming Standard ML of New Jersey. Version 110.0.6, January 10, 2001. Available from: http://www.cs.cornell.edu/riccardo/smlnj.html.

[62] Bjorn De Sutter, Frank Tip, and Julian Dolby. Customization of Java Library Classes using Type Constraints and Profile Information. In *Proceedings of ECOOP 2004*, pages 585 – 609, Oslo, Norway, 2004.

[63] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison Wesley, 1999.

[64] Frank Tip, Adam Kiezun, and Dirk Baumer. Refactoring for Generalization using Type Constraints. In *Proceedings of OOPSLA 2003*, pages 13–26, Anaheim, California, USA, 2003.

[65] Jeffrey D. Ullman. *Elements of ML Programming.* Prentice Hall, Upper Saddle River, NJ, USA, 1998.

[66] M. G. J. van den Brand, P. Klint, and C. Verhoef. Reverse Engineering and System Renovation – An Annotated Bibliography. *ACM SIGSOFT Software Engineering Notes*, 22(1):57–68, Jan 1997.

[67] Mark G. J. van den Brand, Paul Klint, and Jurgen J. Vinju. Term rewriting with traversal functions. *ACM Trans. Softw. Eng. Methodol.*, 12(2):152–190, 2003.

[68] Jonne van Wijngaarden and Eelco Visser. Program Transformation Mechanics. A Classification of Mechanisms for Program Transformation with a Survey of Existing Transformation Systems. Technical Report UU-CS-2003-048, Institute of Information and Computing Sciences, Utrecht University, 2003.

[69] E. Visser, Z. e. A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.

[70] Eelco Visser. A Survey of Strategies in Rule-Based Program Transformation Systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005. Special issue on Reduction Strategies in Rewriting and Programming.

[71] W3C. W3C XML Transformation Standards: c2010. Available from: http://www.w3.org/standards/xml/transformation.

[72] G. L. Wickstrom, J. Davis, S. E. Morrison, S. Roach, and V. L. Winter. The SSP: An Example of High-Assurance System Engineering. In *HASE 2004: The* 8^{th} *IEEE International Symposium on High Assurance Systems Engineering*, pages 167–177, Tampa, Florida, United States, 2004. IEEE.

[73] V. L. Winter and J. M. Boyle. Proving Refinement Transformations for Deriving High-Assurance Software. In *Proceedings of the IEEE High-Assurance Systems Engineering Workshop*, pages 68–77, 1996.

[74] V. L. Winter, S. Roach, and G. Wickstrom. Transformation-oriented Programming: A development methodology for high assurance software. In M. Zelkowitz, editor, *Advances in Computers: Highly Dependable Software*, volume 58, pages 47 – 116. Academic Press, 2003.

[75] Victor Winter. Strategy application, observability, and the choice combinator. Technical Report SAND2004-0871, Sandia National Laboratories, March 2004.

[76] Victor Winter and Jason Beranek. Program Transformation Using HATS 1.84. In Ralf Lämmel, Joăo Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 4143 of *LNCS*, pages 378–396, 2006.

[77] Victor Winter and Azamat Mametjanov. Generative Programming Techniques for Java Library Migration. In *Proceedings of the Sixth International Conference on Generative Programming and Component Engineering (GPCE'07)*, Salzburg, Austria, October 2007. ACM Press.

[78] Victor Winter, Azamat Mametjanov, Steven Morrison, James McCoy, and Gregory Wickstrom. Transformation-based Library Adaptation for Embedded Systems. In *Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium (HASE'07)*, Dallas, TX, USA, Nov 2007.

[79] Victor Winter and Mahadevan Subramaniam. The Transient Combinator, Higher-order Strategies, and the Distributed Data Problem. *Science of Computer Programming (Special Issue on Program Transformation)*, 52:165–212, 2004.

[80] Victor L. Winter. Program Transformation: What, How, Why? In *Encyclopedia of Computer Science and Engineering*. Wiley and Sons, 2006.

[81] Victor L. Winter. Stack-based Strategic Control. In *Preproceedings of the Seventh International Workshop on Reduction Strategies in Rewriting and Programming (WRS'07)*, June 2007.

[82] Victor L. Winter, Christopher Scalzo, Arpit Jain, Brent Kucera, and Azamat Mametjanov. Comprehension of generative techniques. In *Software Transformation Systems Workshop (STS)*, 2006.

[83] V.L. Winter. Strategy Construction in the Higher-Order Framework of TL. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 124, 2004.

[84] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 249–257, Montreal, Quebec, Canada, 1998.