Automatic Analysis, Verification, and Generation of Functional Programs
(関数プログラムの自動解析・検証・生成)

Mizuhito Ogawa

小川 瑞史

Japan Science and Technology Corporation, PRESTO

科学技術振興事業団 さきがけ研究21

A thesis submitted in partial fulfillment of
the requirements for the degree of Doctor of Science
in Information Science at the University of Tokyo

March 2002

Committee

Masami Hagiya	Professor (Cheif Examiner)
Ryu Hasegawa	Associate Professor
Shinichi Morishita	Associate Professor
Hiroyuki Sato	Associate Professor
Masato Takeichi	Professor

(in alphabetical order)

Acknowledgements

First of all, I would like to thank my supervisor, Masami Hagiya, for his great guidance and encouragement, which have been invaluable to the research in this thesis and beyond.

I also feel deeply indebted to Satoshi Ono, his constant encouragement, leading, and advice. He has been patiently to hear and discuss what I am thinking of, and has given me immense guidance from my very early stage as a researcher. I always enjoyed the collaborated work with him, and any of works in this thesis cannot be even started without him.

I could not forget to mention the great and pleasant time to collaborate with Vincent van Oostrom, Zurab Khasidashvili, Ken Mano, Michio Oyamaguchi, Yoshikatsu Ohta, Kunihiro Matsuura, Masato Takeichi, Zhenjiang Hu, and Isao Sasano. Without the discussions with them, most of works in the thesis cannot be done.

I benefited a great deal from stimulating discussions with both CACA seminar and TRS meeting members, especially, Robert Glueck, Aart Middeldorp, Tohru Naoi, Toshiki Sakabe, Masahiko Sakai, Masako Takahashi, Akihiko Takano, Yoshihito Toyama, and Fer-Jan de Vries. They help me understanding important issues and inspiring new ideas related to this research.

Thanks also go to Thomas Arts, Mariangiola Dezani, Peter Dybjer, Yoshihiko Futamura, Ryu Hasegawa, Yasuyoshi Inagaki, Masami Ito, Yukiyoshi Kameyama, Takuya Katayama, Yoshiki Kinoshita, Jan-William Klop, Hidetaka Kondoh, Armin Kühnemann, Ronald van der Meyden, Shinichi Morishita, Jean-Eric PIN, Femke van Raamsdonk, Hiroyuki Sato, and Philip Wadler for discussing about related research through conferences, informal seminars, and private communication.

Many supervisors and colleagues when I worked at NTT, particularly Yoh'ichi Tohkura, Shigeki Goto, Ikuo Takeuchi, Makoto Amamiya, Yasushi Hibino, Ryuzo Hasegawa, Naohisa Takahashi, Masaru Takesue, Hiroshi G. Okuno, Hisao Nojima, Yasunori Harada, Hideki Sakurada, Yasuyuki Tsukada, and Juunosuke Yamada have helped and made my years enjoyable. Now, many of people listed above have moved from NTT, but I recall pleasant days with them and I would not be myself without their friendship and support.

Last but not least, my gratitude and love are to my wife Mikako and my daughter Mizuho for their support, patience, and love.

Mizuhito Ogawa
March 2002

Contents

Chapter 1

Introduction

Functional programs are said to have many nice features, such as clear and transparent semantics, elegant recursive control structures, and flexible recursive data structures. Their relative, LISP, is one of the oldest programming languages. For instance, the first implementation of LISP was reported in 1959, whereas the first implementation of Fortran was reported on 1957 [Sam69].

At that early stage, machine power was poor, and LISP is proved to be several times less efficient than Fortran as a result of its flexible description and powerful expressibility. Thus most scientific computation inclined towards Fortran. Over the last several decades, hardware has made huge progress, but functional programs still remain in the minority with most developers still using C, C++, visual basic, and, recently, Java. Our basic motivation starts from the question: *Is there room for functional programs to survive*?

Most applications of functional programs are program manipulations, which are typically optimization techniques in compilers, especially compliers for the functional language itself. For instance, GHC (Glasgow Haskell Compiler) is written in Haskell itself, and sometimes is used as a test bed for program transformation/analyses of functional programs. An interesting example would be FFTW (Fastest Fourier Transformation in the West) [Fri99], which implements Cooley-Tukey's algorithm and automatically generates optimized C-codes corresponding to the running environment. This generation is mostly based on program transformation implemented by Caml Light.

Recent efforts have resulted in large-scale practical systems. They include:

1. The database querying system Kleisli, which is based on functional query description language CPL (implemented by SML) and is used in several international genom projects [Won00].

2. The Intel-developed hardware verification system Forte, which is implemented by functional language fl [Seg00].

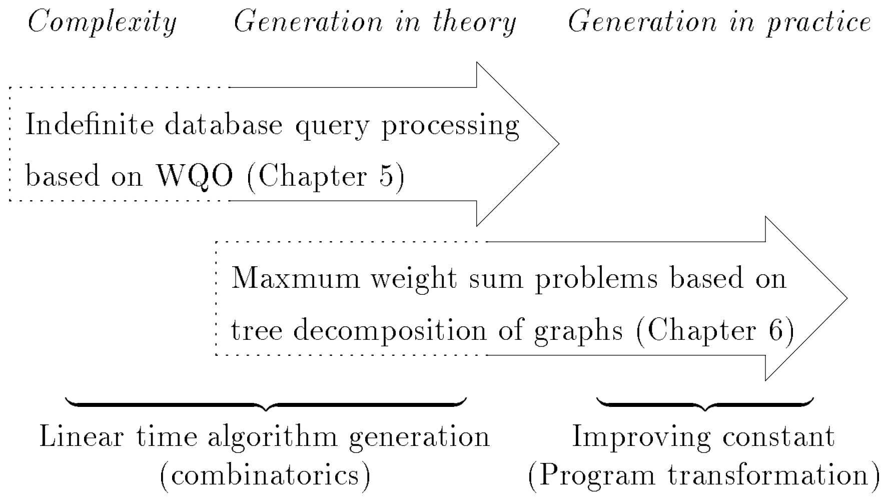
3. Erlang, which is used in telecommunication systems and was developed by Erikson.

The aim of this thesis is to investigate possible theoretical advantages of functional programs as a complement to these empirical results. I believe that such advantages could be obtained by developing automatic or semi-automatic supports to implement correct and efficient programs, such as automatic program analysis, verification, and generation.

The correctness of a program, which is hopefully detected by either analysis or verification, are stratified into termination and partial correctness. For the former, typical automatic termination verification is based on path orderings in simple termination [DJ90] and/or dependency pair analysis [AG00]. However, once these methods fail, solving termination becomes very difficult. A sufficient condition of the property called *uniform normalization*, which reduces termination (strong normalization) to a much easier weak normalization, will be presented in Part I. This will be useful in manually proving the termination. For the latter, automatic supports have been widely investigated as compile time analysis/verification based on abstract interpretation [CC77, Bur91, NNH99], type-based analysis [HLM94, LP00], model checking [Ste91, Sch98], etc. Techniques presented in this thesis are mostly based on abstract interpretation, and will appear in Part II.

The conventional direction above is to automatically debug/optimize hand-coded programs at compile time. Another direction would be to automatically generate an efficient program from specifications, whose correctness is guaranteed by nature.

Such automatic generation (or synthesis) has been investigated since the late 70's, especially in connection with program extraction from constructive proofs. Unfortunately, this direction has been less successful compared to the progress in compilers. This comes from the high cost of both describing specifications (which is often more complex than coding itself) and strong human guidance on heuristics. Of course, critical systems and commonly used kernels are worth the cost, but we also hope light-weight program generation even at the cost of restricting the class of problems. In the approach of this thesis, combinatorics theory, especially well-quasi-orders (WQO) [FL88] and tree decomposition of graphs [Var00, Flu01], will be applied.


Complexity
Generation in theory
Generation in practice
Indefinite database query processing based on WQO (Chapter 5)
Maxmum weight sum problems based on tree decomposition of graphs (Chapter 6)
Linear time algorithm generation (combinatorics)
Improving constant (Program transformation)

In both cases, there would be three stages: proving the existence of a linear time algorithm (estimating complexity), automatic generation in theory, and automatic generation in practice. At each stage, there are gaps; even if the existence of a linear time algorithm has been known, sometimes it is very hard to find an actual algorithm, and even if a linear time algorithm is automatically generated, sometimes the constant is huge and practically useless.

One possible bridge between the first and second stages is the method by WQO. In Chapter 5, the automatic generation of linear-time programs based on WQO is presented. The target example is query processing on indefinite databases, which was posed as an open problem in [van97]. The possibility of reducing the constant by *fold-unfold* program transformation techniques of functional programs will be also discussed.

The possible bridge between the second and third stages is the combination of the tree decomposition and the program calculational techniques of functional programs. In Chapter 6, automatic linear time algorithm generation of maximum weight sum problems based on tree decomposition of graphs will be discussed. There has been lots of works on this subject [BLW87, Cou90, BPT92], but known methods generate a linear time algorithm with huge constant, say, the tower of exponentials. The proposal how to drastically reduce the constant by *fusion / tuppling* transformation will be presented [SHTO00, SHTO02]. Currently, the experience is restricted to only simple cases. To extend the range, we need both to clarify the algebraic structure of construction of more complex graphs and to extend program calculation to non-initial algebras. As the first step, the complete axiomatization of the algebraic construction of graphs in [APS90] will be presented in Section 6.2.

Finally, Chapter 7 discusses future work along with the positioning of the contributions. The work is ongoing, and this thesis presents the present status. In what follows, the contributions of this thesis are presented by way of overviewing each technique.

1.1 Rewriting techniques for proving termination

Several useful techniques of automatic termination verification in rewriting theory have been proposed. Two well-known complementary techniques are simplification orderings [Der82, DF85, DJ90], and dependency pair analysis [AG97, AG00]. Simplification orderings (such as lexicographic path ordering) are quite powerful tools; for instance, the termination of the Ackerman function, which is beyond the class of primitive recursive functions, can be automatically detected. However, once simplification orderings fail, problems become very difficult. Dependency pairs sometimes restore the situation. Simplification orderings alone require that every term be suitably ordered, but the dependency-pair method enables us to restrict our attention to dependency pairs only, which are syntactically computed from rewrite rules. Of course, termination in general is undecidable, so there always remain harder problems. For instance, termination of typed lambda calculus [Bar84, P.92] and explicit termination [Bon00, ACCL91] are typical. Semantic labelling can provide guidance for proving termination manually [Zan95]; however, this method can prove termination of every terminating system. This is too strong in some sense and sometimes one can find suitable semantic labelling only after one has found the termination proof.

Uniform normalization comes from the observation that even if strong normalization (termination) is hard to prove, weak normalization (i.e., existence of a terminating rewrite sequence for each term) would be easy to prove. Simply typed lambda calculus is a such case. Uniform normalization guarantees the equivalence of weak normalization and strong normalization. More precisely, a term t is uniformly normalizing (hereafter, UN for short) if either it does not have any normal form (i.e., t is not weakly normalizing), or all reductions starting from t are finite (i.e., t is strongly normalizing). If every term is UN then the rewrite system is UN. Thus, once uniform normalization is shown, the proof of strong normalization is reduced to a much easier weak normalization. An example of uniform normalization is Church's *Conservation Theorem* for λ_I-calculus [Chu41], and recently a useful UN subclass of λ-terms has been identified in [MNS99].

Chapter 2 presents the simple sufficient condition for uniform normalization of orthogonal higher-order rewrite systems. Orthogonality is quite strong restriction from rewriting point of view, but since orthogonal rewrite systems can describe every partial recursive function orthogonality provides enough rich class from programming point of view.

A key observation of the criteria is that a rewrite system is UN if and only if each reduction step is perpetual and if and only if each redex is perpetual. A perpetual step is a reduction step that retains the possibility of infinite reductions. A perpetual redex is a redex that, when put into an arbitrary context, yields a perpetual step. This observation reduces the uniform normalization to the construction of a perpetual reduction strategy. Then, existing criteria below for the perpetuality to orthogonal higher-order rewrite systems are generalized and refined.

- Folklore lemma that for orthogonal term rewriting systems a reduction step that does not erase any argument of a redex possessing an infinite reduction is perpetual [Klo92].
- *Conservation Theorem* [BBKH76, Bar84], i.e., β_I-redexes in λ-calculus are perpetual.
- A necessary and sufficient criterion for the perpetuality of β_K-redexes [BK82].

As a framework for higher-order rewrite systems, there are two major choices: one *with* or one *without* metavariables. Typical examples of the former are combinatory reduction systems [Klo80, KvOvR93] and expression reduction systems [KvO95a] (which are almost equivalent), and those of the latter are pattern rewrite systems [MN98] and higher-order rewriting [VO94]. Instead of metavariables, the latter uses a substitution calculus that pushes out the *arguments* from a redex when matching is done with the left-hand side of a rule. For instance, the *map* function is expressed as

$$\begin{array}{lcll} map(F, x : xs) & \rightarrow & (F\ x) : map(F, xs) & \text{(with metavariables)} \\ \lambda F\ x\ xs.map(F, x : xs) & \rightarrow & \lambda F\ x\ xs.(F\ x) : map(F, xs) & \text{(without metavariables)} \end{array}$$

and the latter reduces $map(1+, 3 : [4, 5])$ as

$$\begin{array}{rcl} map(1+, 3 : [4,5]) & \Rightarrow^* & (\lambda F\ x\ xs.map(F, x : xs))\ 1+\ \ 3\ [4,5] \\ & \rightarrow & (\lambda F\ x\ xs.(F\ x) : map(F, xs))\ 1+\ \ 3\ [4,5] \\ & \Leftarrow^* & (1 + (3)) : map(1+, [4,5]) \end{array}$$

where $\Rightarrow$ is a reduction of the substitution calculus (usually, $\lambda_{\beta\bar{\eta}}$ is used).

The choice here is the former; *Context-sensitive Conditional Expression Reduction Systems* (CCERSs) are adopted and a concept of orthogonality, which implies confluence, is defined. In particular, several important λ-calculi and their extensions and restrictions can be naturally embedded into orthogonal CCERSs. Then, a perpetual reduction strategy that enables one to construct *minimal* (w.r.t. Lévy's *permutation* ordering on reductions) infinite reductions in orthogonal fully-extended CCERSs, is introduced.

Using the properties of the minimal perpetual strategy,

1. the perpetuality of any reduction step that does not erase *potentially infinite* arguments, which are arguments that may become, via substitution, infinite after a number of *outside* steps, and

2. the perpetuality (in every context) of any *safe* redex, which is a redex whose substitution instances may discard infinite arguments only when the corresponding contracta remain infinite.

are proved.

We prove both these perpetuality criteria for orthogonal fully-extended CCERSs and then specialize and apply them to restricted λ-calculi. These criteria cover most of the known results on uniform normalization [Klo92, Chu41, BBKH76, Bar84, BK82, DG93, HL93, Len97a]. An exception is [BI94], which treats weakly orthogonal rewrite systems.

Note that even the Conservation theorem fails for orthogonal higher-order rewriting [VO94, MN98] in general. Instead, [KOvO01b] showed that the similar perpetuality criteria hold for weakly orthogonal second-order rewriting, and cover the result in [BI94].

For consistency of rewriting systems, the author also proved Chew's theorem [MO01], whose proof is a long standing problem since 1981 [Che81]. Chew's theorem gives a sufficient condition for the unique normal form property, which is closely related to consistency. This result is not included in this thesis.

1.2 Analyses and verification of hand-coded programs based on abstract interpretation

Abstract interpretation firstly appeared in the seminal paper [CC77] and introduced into the functional world in [Myc80]. The basic idea is to interpret analyses as program executions on finitely abstracted domains. If the finite domains properly reflect the property to be detected, then the executions for all possible inputs will give some information regarding the property. If abstract domains are designed to be finite, such executions terminate. A typical example is an analysis of the sign of the addition, i.e., determining whether the output is positive or negative: with the abstract domain consisting of two elements $+$ and $-$, the result of the addition of $+$ and $+$ is $+$, that of $-$ and $-$ is $-$, but the result of the addition of $+$ and $-$ cannot be decided and we set it as "$+$ or $-$" ($\{+,-\}$). This ambiguity comes from approximation, and this is an inevitable cost for termination of analyses. The more complex domain will detect more detailed information, but of course the design of abstract domains is always carried out under a trade-off between computational feasibility and analyzing power. For instance, if one would like to further analyze what happens to the result of multiplication with 0, the abstract domain must be enlarged to a three-point domain $\{+, -, 0\}$ from the analysis above. Then one will find that the multiplication with 0 returns 0.

There are two possible directions in abstract interpretation. One is to pursue efficiency to analyze relatively simple properties of large-scale programs. The other way is to pursue analyzing power to analyze difficult properties of relatively small programs. The direction of this thesis is the latter.

In Chapter 3, an abstract interpretation based on a domain abstraction is characterized by a quadruplet, consisting of a pair composed of abstraction and concretization, direction, power domain construction (i.e., a pair composed of a quasi order and a closure function), and a representative function. This quadruplet is called a HOMomorphic Transformer (HOMT), and when the composition of HOMTs are reduced is analyzed. This enables us to compare and evaluate analyses/verifications, which are independently proposed. For instance, known strictness analyses are compared in this framework [OO91b].

For instance, the example above, the analyses of the sign of the addition in a two-point abstract domain $\{+, -\}$, is formalized as a HOMT as follows:

- the domain abstraction *abs* and the domain concretization $conc (= abs^{-1})$.

$$abs : \begin{cases} 0, 1, 2, \cdots & \rightarrow & + \\ -1, -2, \cdots & \rightarrow & - \\ \perp & \rightarrow & \perp \end{cases} \qquad conc : \begin{cases} + & \rightarrow & \{0, 1, 2, \cdots, \perp\} \\ - & \rightarrow & \{-1, -2, \cdots, \perp\} \\ \perp & \rightarrow & \{\perp\} \end{cases}$$

- the direction: *forward.*

- the closure $cl(X) = X \cup \{\perp\}$ and the quasi order $X \sqsubseteq Y$ if and only if $cl(X) \subseteq cl(Y)$.

- the representative function (over the abstract power set): $rep : X \rightarrow X \cup \{\perp\}$.

The forward analysis detects the properties by computing all possible computations on the abstract domain, while the backward analysis detects the properties in a demand-driven manner. In the case of the analysis of the sign, the backward analysis detects that positive output requires that a pair of inputs is not $(-, -)$.

The computation on the abstract domain is executed over its power domain, and the quasi order expresses the definedness over abstract power domains. This simple example does not show why the representative function is needed, but this will be clarified when the composition of HOMTs is introduced in Chapter 3.

Next, Chapter 3 presents a computation path analysis (CPA) [小86, OO91b] and its applications. CPA detects all possible computation paths, which expresses demands propagation patterns. Consider $if(x, y, z)$; a strictness analysis only detects that x is strict, but CPA detects that demands are propagated to either $\{x, y\}$ or $\{x, z\}$.

Comparing CPA with various strictness analyses and projection analysis [WH87, DW90, DW91], the formalization by HOMTs shows that it is more powerful than strictness analyses and has equivalent power to projection analysis. For instance, CPA detects *head strictness*, where head strictness cannot be detected by conventional strictness analyses (other than [Hun91]). Head strictness means the synchronous evaluation of leaves (the car part) to the evaluation of the spine (the cdr part), and the typical example of head strictness is a function $find0(x)$, which returns *true* if x contains 0, and *false*, otherwise.

These analyses are useful for debugging and optimizing demand-driven programs, especially those with recursive data structures, such as lists and streams. A non-strict cons delays the evaluation of its argument parts, typically the tail part. Such delay operations require closure constructions, which is usually expensive.

First possibility is to manually specify the use of strict/non-strict cons by his/her responsibility, such as in [Hen80] and delay/force in Scheme [SF89]. This may be error prone and would easily produce divergence. To remedy the situation, anomaly detection [小88], which detects irrelevant objects and diverged objects, will be introduced in Section 3.3. For instance, CPA detects anomalies such as

- an irrelevant object y in $foo(x, y) = if\ x == 0\ then\ 1\ else\ foo(x-1, foo(y, x))$, and
- a diverged object $intseq(n) = cons(n, instseq(n+1))$ where *cons* is strict.

Another possibility is to automatically optimize a demand-driven programs, such as Haskell or Clean, by using so-called *call-by-need to call-by-value transformation*. CPA also works for this direction [OTA86], but there have been widely investigated [Bur91, NNH99, PJ87, AH87] and applied in compilers, such as GHC (Glasgow Haskell Compiler, see `http://haskell.org`). Thus, we prefer not to put more words on it.

The last section in Chapter 3 presents the implementation of CPA. Various CPAs are constructed by designing modes, that is, designing abstract domains [小91]. It also presents an automatic analyzer generator, which accepts the definition of modes as a finite lattice.

Chapter 4 introduces a more powerful application of abstract interpretation over recursively defined domains.[1] That is, it describes the verification of binary properties [Oga99], whereas Chapter 3 shows the analyses of monadic properties [小91]. Analysis and verification are complementary; they are two sides of the same coin. Abstract interpretation needs an approximation; analysis approximates from the necessary condition side and verification approximates from the sufficient condition side.

This verification was firstly proposed in [LM95], and [Oga99] reconstructed and extended it based on a backward abstract interpretation. Two new techniques are introduced to abstract interpretation:

1. lazy abstract domain construction, and
2. inductive predicate constructors.

[1]Here, only lists are treated, but this is easily extended to recursive data structures of regular types [BdM96].

From the termination requirement of abstract interpretation, abstract domains must be finite. If an abstract domain can be generated dependent on an input program to be analyzed, we would expect to be able to maximize the analyzing ability. The verification is done in a backward manner, and the abstract domain is lazily created as a set of formulae constructed from the property to be detected and function/variable symbols appear in an input program.

For instance, if a list $x : xs$ satisfies $\nabla geq\ x : xs$ (which means $x : xs$ is a decreasing list), this is decomposed to $\forall leq^x\ xs$ and $\nabla geq\ xs$, where $\forall leq^x\ xs$ means that each element in xs satisfies a newly introduced predicate leq^x, i.e., *less-than-equal* to x. Note that $\forall leq^x$ is constructed from the basic predicate leq, the variable name x (which appears in a program), and the inductive predicate constructor $\forall$. Since prepared inductive predicate constructors always lift a predicate to that of more complex types, only finitely many nests of predicate constructors are possible, considering the strongly typed nature of a program, i.e., typing of a program gives an upper bound. Further, since the number of variable symbols that appear in a program is finite, the number of newly constructed predicates (i.e., elements of an abstract domain) will be bound. This guarantees the termination of the verification.

The verification presented in Chapter 4 also allow the binary predicate to represent the property. Difficulties arise when a variable x needs to satisfy $leq(x, y)$ with a local variable y. But when the verification algorithm traces the program, the tracing point will eventually leave the scope of y. In such a case, $leq(x, y)$ must be approximated to the formula without y. This can be done by observing how y is locally defined. For instance, if y comes as $let\ y : ys = xs\ in\ \cdots$, then $leq(x, y)$ is approximated to $\forall_r leq(x, xs)$, which means that x is less-than-equal to each element in xs. Here $\forall_r$ is an example of an inductive predicate constructor.

Typical examples of verification are the properties of various sorting programs, such as decreasingness (i.e., the resulting list is in decreasing order) and weak preservation. The former is represented as

$$\mathbf{true} \rightarrow \nabla geq\ (sort\ X),$$

which is interpreted as the **true** assumption (i.e., no assumptions) on an input resulting in $\nabla geq\ (sort\ X)$ (i.e., $sort\ X$ is a decreasing list). The latter, weak preservation, means that any element appearing in an input also appears in an output of *sort*, and vice versa. This is expressed as

$$\mathbf{true} \rightarrow (\forall_l \exists_r equal \wedge \forall_r \exists_l equal)^X (sort\ X).$$

At last, This part is the new termination criteria for abstract interpretation concluded Part 2. Analyses in Chapter 3 use prefixed finite abstract domains, and verification in Chapter 4 uses generated finite abstract domains (dependent to an input program to verify). With the aids of *better-quasi-order* (BQO), we can extend the termination criteria beyond finiteness. The main statement is, roughly speaking, *if an abstract domain is better-quasi-ordered then a backward abstract interpretation terminates.* Although no new applications have been found, this criteria gives a proper extension.

1.3 Automatic generation of efficient programs based on combinatorics

Recent developments in program transformation, such as partial evaluation and program calculation, have enabled us to derive efficient algorithms from simple specifications/programs. However, there is trade-off between automatic derivation and increased speed, i.e., most transformations that decrease complexity require human guidance.

One possible way to avoid the need for such guidance is to apply mathematics. In Chapter 5, WQO is applied, and in Chapter 6, tree decomposition of graphs is applied.

In Chapter 5, a linear time algorithm generation based on WQO techniques is presented. WQO guarantees the finiteness of the minimal elements of any set. If the set is upward closed, such minimal elements, called *minors*, precisely characterize the set. As a result, the membership problem associated with an upward closed set is reduced to a comparison with such a finite set of minors. To illustrate, the Graph Minor Theorem states that the embedding relation on finite graphs is a WQO [NP88], and this implies the existence of square time algorithms for a wide range of graph problems [FL88, PR99]. Unfortunately, because if its highly nonconstructive nature, it does not allow us to effectively detect minors. Thus, constructing the actual algorithm is very difficult in many cases.

Chapter 5 will solve such difficulties for the simplest case, i.e., the embedding over words.b The idea is suggested by the result of Ehrenfeucht et al. that *a set L of finite words is regular if and only if L is $\leq$-closed under some monotone well-quasi-order (WQO) $\leq$ over finite words* [EHR83]. This gives the insight that upward closed sets would be described by regular expressions and introducing well-founded orders on such descriptions would give computational contents of Higman's lemma, which states that the embedding relation on finite words is a WQO. Murthy and Russell gave its constructive proof by the such scenario [MR90].

The target example is the automatic generation of linear time disjunctive monadic query processing in an indefinite database on a linearly ordered domain. This problem was first posed by Van der Meyden in [van97], and he showed the existence of a linear time algorithm, but its actual construction has not been reported. Van der Meyden's proof of existence is based on Higman's lemma. Using the techniques of its constructive proof in [MR90], minors for a given disjunctive monadic query can be effectively computed. This will lead to the construction of a linear time algorithm of disjunctive monadic query processing.

To get a feeling for the target problem, let us consider its simpler version. Let x and y be lists, and let $sublst(x, y)$ be a predicate that returns *true* if x is a sublist of y, and returns *false* otherwise. More rigorously, we can write in Haskell:

```
sublst :: [a] -> [a] -> Bool
sublst    []     ys  = true
sublst  x:xs     []  = false
sublst (x:xs) (y:ys) = if x == y then sublst xs ys
                                 else sublst (x:xs) ys
```

Let us consider an easy problem. Fix a list x.

- **Input**: A finite set of lists $\bar{y} = \{y_1, \cdots, y_t\}$.
- **Output**: A decision as to whether any list z with $\wedge_{j=1}^{t} sublst(y_j, z)$ holds $sublst(x, z)$.

This problem can be regarded as a query as follows: We know partial information on events (which exclusively occur), and this partial information is represented as a set of lists y_j's. Then, the question is: *Can we decide whether there exists an event sequence represented as x?* This problem is simply solved by computing $sublst(x, y_j)$ for each y_j, and if some $sublst(x, y_j)$ returns *true*, then it holds; otherwise it does not.

Now we consider two extensions: (a simpler version of) conjunctive query and disjunctive query. The conjunctive query is as follows: fix a finite number of lists, $x_1, \cdots, x_s$.

- **Input**: A finite set of lists $\bar{y} = \{y_1, \cdots, y_t\}$.
- **Output**: A decision as to whether any list z with $\wedge_{j=1}^{t} sublst(y_j, z)$ holds $\wedge_{i=1}^{s} sublst(x_i, z)$.

This is still easy, since this problem is decomposed into a check on each x_i, i.e., whether for each x_i, $sublst(x_i, y_j)$ for some y_j holds.

However, the disjunctive query is much harder. The disjunctive query is formalized as follows: fix finite number of lists, $x_1, \cdots, x_s$.

- **Input**: A finite set of lists $\bar{y} = \{y_1, \cdots, y_t\}$.
- **Output**: A decision as to whether any list z with $\wedge_{j=1}^{t} sublst(y_j, z)$ holds $\vee_{i=1}^{s} sublst(x_i, z)$.

Finding an efficient solution (a linear time algorithm) for this problem is not as easy as it appears. To illustrate, consider $x_1 = [P, Q, R]$, $x_2 = [Q, R, P]$, and $x_3 = [R, P, Q]$. This holds for $y_1 = [P, Q]$, $y_2 = [Q, R]$, and $y_3 = [R, P]$, even though none of the x_i's and y_j's hold $sublst(x_i, y_j)$.

Of course, if one computes every possible combination of z, a decision is possible, but this requires an exponentially greater amount of time. For instance, for lists $y_1, \cdots, y_t$ of lengths $n_1, \cdots, n_s$, the number of combinations is $(n_1 + \cdots + n_t)!/(n_1! \times \cdots \times n_t!)$, which grows exponentially.

The aim is to generate the linear time algorithm for a given disjunctive query. For this purpose, a suitable finite set $\mathcal{M}$ of a finite set of lists, called *minors*, is generated corresponding to x_i's. Namely, for the example $x_1 = [P, Q, R]$, $x_2 = [Q, R, P]$, and $x_3 = [R, P, Q]$ above,

$$\begin{array}{l}
\{[P,Q],[Q,R],[R,P]\}, \{[P,Q,R]\}, \{[Q,R,P]\}, \{[R,P,Q]\}, \\
\{[P,Q,P],[Q,R]\}, \{[Q,R,Q],[R,P]\}, \{[R,P,R],[P,Q]\}, \\
\{[P,R,P],[Q,R]\}, \{[Q,P,Q],[R,P]\}, \{[R,Q,R],[P,Q]\}
\end{array}$$

Then the disjunctive query for input $\bar{y} = \{y_1, \cdots, y_t\}$ is reduced as to whether there exists a minor $\bar{m}$ in $\mathcal{M}$ such that for each $m \in \bar{m}$ there exists y_j satisfying $sublst(m, y_j)$.

The finiteness of minors is guaranteed by Higman's lemma. When generating minors, the most difficult aspect is knowing whether all have been found. To do this, the constructive proof of Higman's lemma based on regular expressions is applied.

Chapter 5 also gives the extension of the result of Ehrenfeucht et al. that *a set L of finite words is regular if and only if L is $\leq$-closed under some monotone well-quasi-order $\leq$ over finite words* [EHR83] to regular ω-languages. That is,

1. an ω-language L is regular if and only if L is $\preceq$-closed under a *periodic* extension $\preceq$ of some monotone WQO over finite words, and
2. an ω-language L is regular if and only if L is $\preceq$-closed under a WQO $\preceq$ over ω-words which is a *continuous* extension of some monotone WQO over finite words.

In Chapter 6, a *practical* linear time algorithm generation based on tree decomposition is discussed. Graphs are flexible data structures but are also sources of NP-hardness. Many graph algorithms have been investigated individually. As a researcher of programming, I am inclined to want to design suitable programming languages that can manage graphs, or to give methods of systematic derivation and/or automatic generation of graph algorithms.

As an example of systematic derivation of a graph algorithm, the Dijkstra algorithm, which is the most efficient one known for computing the shortest path in a weighted graph with complexity $O(e\ log\ n)$ (where e is the number of edges and n is the number of vertices), can be derived from the naive functional program by program calculational techniques [篠00]. This is nice, but strong derivation that may decrease complexity often requires strong human guidance.

On the other hand, by using the tree decomposition techniques for graphs with bounded tree width [RS86, RS95, ACPS93], there have been investigated on automatic linear-time algorithm generation for problems specified by monadic second-order formulae [BLW87, Cou90]. The class of problems is extended to maximum weight sum problems in which constraints are specified by monadic second-order formulae [BPT92]. Typical examples are the shortest path problem (specified by the reachability constraint), the party planning problem [CLR90, BdM96] (specified by the independence constraint), and the maximum segment sum problem [Gri90, Ben84] (specified by the connected constraint).[2] The key is the dynamic programming technique that traces an algebraic construction of a graph, which is illustrated in Fig. 1.1. This is excellent in theory, but is still infeasible in practice, because of the huge constants of generated linear time algorithms. Such constants easily explode to the tower of exponentials.

The approach in this thesis is; starting from specification by the restricted class of simple recursive functions, called *finite mutumorphism*, instead of that by formulae, fusion/tuppling program transformation of functional programs would drastically reduce the constant [SHTO00]. Sometimes this method supplies programs that are as fast as or faster (both in theory and practice) than hand-coded ones [FMMT96b, BRS99, SHT01, SHTO02]. Currently, known examples are restricted to the cases corresponding to simple

[2]See the list of monadic second order definable properties in [Cou90].

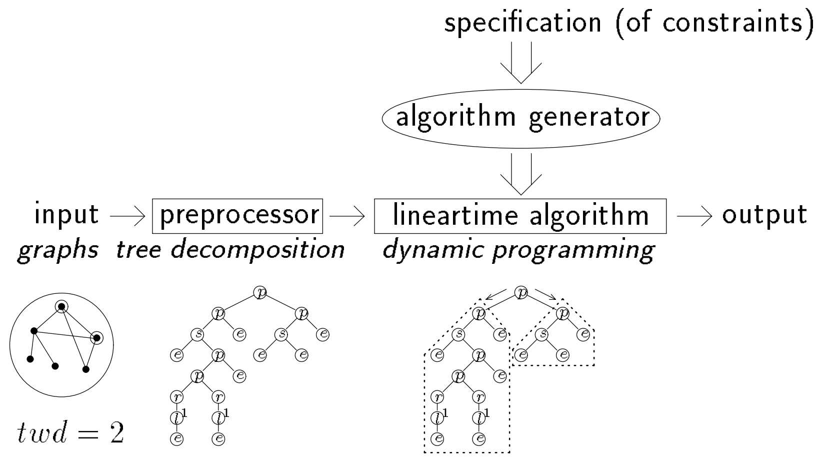


Figure 1.1: Linear time algorithm generation for graphs with bounded tree width

graphs, such as words. Chapter 6 explores the possibility of extension to more complex graphs by analyzing algebraic structure of the algebraic construction of graphs.

First, hand-coded programs and automatically generated ones of the same problems called *maximum segment sum* problem (MSS) [Ben84, BPT92, SHTO00] and k-*maximum segment sum* problem (k-MSS) [BRS99, SHTO02] are presented. MSS receives a finite sequence of (possibly negative) integers and computes a contiguous subsequence such that the sum of elements in the subsequence is maximum. MSS is simple, but has important applications for data mining [FMMT96b]. Surprisingly, MSS was believed to be $O(n\ logn)$ in 70's, and the first linear time algorithm was presented in [Ben84]. The method in this thesis automatically generates a linear time algorithm from the simple functional specification. It is worth mentioning that for the extension of MSS, called k-MSS, a linear time algorithm was quite recently given in [BRS99]. A different but similarly efficient linear time algorithm can be automatically generated by our method [SHTO02].

To explore the possibility of extension to complex structures, such as to beat the cubic bottleneck in control flow analyses [HM97, Rep98], we face the problem that the algebraic construction of graphs (such as tree decomposition) is *not* initial, whereas most of program calculational techniques assume an initial algebra. This problem was already found in the derivation of the Dijkstra algorithm on the shortest path problem [篠00]. Thus, we need the extensions of fusion/tuppling program transformational techniques to non-initial algebra, which were suggested in [Fok96]. As the first step, the complete axiomatization of algebraic construction of graphs is presented. Another complete axiomatization of the different algebraic construction of graphs was already presented in [BC87]; however, their construction almost squarely grows in size. This is not suitable for a linear time generation. Instead, the complete axiomatization to the algebraic construction proposed in [ACPS93], in which the size grows linearly, is presented.

At the moment, whether some finite fragment of this infinite axiomatization is complete for graphs with bounded tree-width is not clear; but I strongly believe such completeness holds and this will be the next step.

Part I

Proving termination of programs

Chapter 2

Uniform normalization of orthogonal rewrite systems

The main objective of this chapter is to study sufficient conditions for *uniform normalization* [KOvO01a]. Here a term t is uniformly normalizing, UN for short, if either it does not have any normal form (t is not weakly normalizing), or all reductions starting from t are finite, (t is strongly normalizing). We study UN for both first- and higher-order orthogonal term rewrite systems, where a rewrite system is said to be UN if each of its terms is so.

Interest in the criteria for UN arises, for example, in the proofs of strong normalization of typed λ-calculi, since these criteria are related to the work on reducing strong normalization proofs to proving weak normalization [Ned73, Klo80, Kar85, DV87, DG93, Kha94a, KW95b, KW95a, Sør97b, Xi97, MNS99]. Furthermore, the question: 'Which classes of terms are UN ?' is posed by Böhm and Intrigila [BI94] in connection with finding UN solutions to fixed point equations, and with the representability of partial recursive functions by UN terms only, in the λ-calculus.[1] A useful UN subclass of λ-terms has recently been identified by Møller Neergaard and Sørensen [MNS99].

Let us call a term t an ∞-*term* if it has an infinite reduction. Furthermore, we call a reduction step $t \to s$ and the corresponding contracted redex-occurrence *perpetual* if s is an ∞-term if t is so. A redex is called perpetual if its occurrence in every context (and the corresponding reduction step) is perpetual. It is easy to see that a rewriting system is UN iff all of its reduction steps are perpetual iff all of its redexes are perpetual. Studying uniform normalization therefore reduces to studying the perpetuality of redexes and reduction steps, which has been studied quite extensively. The classical results in this direction are Church's *Conservation Theorem for the λ_I-calculus* [Chu41], stating that the λ_I-calculus is UN, and the *Conservation Theorem* (for the λ_K-calculus) due to Barendregt, Bergstra, Klop and Volken [BBKH76, Bar84], stating that β_I-redexes are perpetual in the λ-calculus. Bergstra and Klop [BK82] gave a necessary and sufficient criterion for the perpetuality of β_K-redexes. Klop [Klo80] generalized Church's Theorem to non-erasing orthogonal Combinatory Reduction Systems (CRSs) by showing that those systems are UN, and Khasidashvili [Kha94a, Kha01] generalized the Conser-

[1] Uniform normalization is called *strong normalization* in [BI94].

vation Theorem to orthogonal Expression Reduction Systems (ERSs) by proving that all non-erasing redexes are perpetual in orthogonal fully-extended ERSs.[2]

For orthogonal Term Rewriting Systems (TRSs), Klop [Klo92] obtained a very powerful perpetuality criterion in terms of *critical* steps (or critical redex-occurrences). These are steps that are not perpetual, i.e., they reduce ∞-terms to SN terms. Klop showed that any critical step (contracting a redex-occurrence u) must erase an argument of u possessing an infinite reduction. This is not true for orthogonal higher-order rewrite systems, because substitutions (from the outside) into the arguments of u may occur during rewrite steps and such substitutions may turn a SN argument of u into an ∞-term. However, we show that (1) a critical step $t \xrightarrow{u} s$ must necessarily erase a *potentially infinite* argument, i.e., an argument that would become an ∞-(sub)term after a number of (*passive*, i.e., performed in the context of u) steps in t. From this we derive another criterion stating (2) perpetuality of *safe* redexes (in every context), which is similar to the perpetuality criterion for β_K-redexes [BK82]. These two criteria are the main results of this chapter, and we will demonstrate their usefulness in applications.

To unify our results with the ones already in the literature for different orthogonal rewrite systems, we first introduce a framework of *Context-sensitive Conditional Expression Reduction Systems* (CCERSs). This framework provides a format for higher-order rewriting which extends ERSs [Kha92] by allowing restrictions on term formation, on arguments of redexes, and on the contexts in which the redexes can be contracted. Various interesting typed λ-calculi (such as the simply typed λ-calculus [P.92], its extension with pairing [TS96], and system **F** [P.92]) and λ–calculi with specific reduction strategies (such as the call-by-value λ-calculus [Plo75]) can be directly encoded as CCERSs (see also [KvOvR93]). After demonstrating the expressiveness of CCERSs, we will focus our attention on orthogonal CCERSs, present a concept of orthogonality for CCERSs, and prove the standard results for orthogonal CCERSs (the Finite Developments Theorem [FD], confluence, etc.). Further, by necessity, we will restrict our attention to *fully-extended* orthogonal CCERSs; roughly, in fully-extended CCERSs, an erasing step cannot turn a non-admissible redex into an admissible one.

To prove our perpetuality criteria, we will first generalize, from term rewriting and the λ-calculus to orthogonal fully-extended CCERSs, the *constricting* perpetual strategies discovered independently by Plaisted [Pla93], Gramlich [Gra96], Sørensen [Sør98], and Melliès [Mel96]. These strategies specify a construction of infinite reductions (whenever possible) such that all steps are performed in some smallest ∞-subterm. Our strategy is slightly more general than the constricting ones (i.e., it specifies a set of redexes from which any one can be selected for contraction), and can be restricted so that resulting reduction sequences become constricting. The restricted strategy allows for simple and concise proofs of our perpetuality criteria. We will also show that constricting perpetual reductions are minimal w.r.t. Lévy's *permutation ordering* on reductions in orthogonal rewriting systems [Lév80, HL91].

Even though our criteria are simple and intuitive, they are strong tools in proving strong normalization from weak normalization in orthogonal (typed or type-free) rewrite

[2]The restriction to full extendedness was missing in [Kha94a]; full extendedness simply means that no rules are subject to occur conditions like the one in the η-rule.

systems. We will show that all known related criteria [Chu41, BBKH76, BK82, Klo80, Klo92, Kha94a], except the one in [HL99], can be obtained as special cases. We will also demonstrate that uniform normalization for a number of variations of β-reduction (most of which cannot be derived from previously known perpetuality criteria) [Plo75, DG93, BI94, HL93, Len97a] is an immediate consequence of our criteria. ERSs are similar to the Klop's CRSs [Klo80] and we claim that all our results are valid for orthogonal fully-extended CRSs as well (see [VR96] for a detailed comparison of various forms of higher-order rewriting). We will demonstrate, however, that our results cannot be extended to higher-order rewriting systems where function variables can be bound [Wol93, Nip93, VOvR94], since already the Conservation Theorem fails for these systems.

The chapter is organized as follows: In Section 2.1, we introduce CCERSs and show how several rewrite and transition systems can be encoded as CCERSs. In Section 2.2, we prove some standard results for orthogonal CCERSs. In Section 2.3, we study properties of an extension of existing constricting perpetual strategies, and in Section 2.4, we use these properties to obtain our perpetuality criteria for orthogonal fully-extended CCERSs. Section 2.5 gives a number of applications, and Section 2.6 concludes the chapter.

2.1 Context-sensitive Conditional ERSs

A pair $(A, \to)$ of a set A and a relation $\to$ on A is an abstract rewrite system (ARS). A transitive closure, transitive reflexive closure, and transitive reflexive symmetric closure of $\to$ are denoted by $\to^+$, $\to^*$, and $\leftrightarrow^*$, respectively. If an element $a \in A$ has no elements $a' \in A$ such that $a \to a'$, we say a a normal form. A set of normal forms is denoted by NF. An ARS $(A, \to)$ is weakly normalizing if any element $a \in A$ has a finite rewrite sequence $a \to a_1 \to \cdots \to a_n$ such that $a_n \in NF$. An ARS $(A, \to)$ is terminating (or, strongly normalizing, **SN** for short) if there are no infinite rewrite sequences $a \to a_1 \to \cdots \to a_n \to \cdots$. An ARS $(A, \to)$ is locally confluent if for any pair $a, a' \in A$ with $a \leftarrow \cdot \to a'$ there exists $a'' \in A$ such that $a \to^* a''$ and $a' \to^* a''$. An ARS $(A, \to)$ is confluent if for any pair $a, a' \in A$ with $a \leftarrow^* \cdot \to^* a'$ there exists $a'' \in A$ such that $a \to^* a''$ and $a' \to^* a''$.

Lemma 2.1 *A terminating and locally confluent ARS is confluent.*

A term rewriting system (TRS) is a pair consisting of an alphabet and a set of rewrite rules. A rewrite rule is a pair of terms denoted by $l \to r$ satisfying two conditions: (1) l is not a variable and (2) $V(l) \supseteq V(r)$. We call l and r the left-hand side (LHS) and the right-hand side (RHS) of $l \to r$, respectively. The alphabet is used *freely* to generate the terms and the rewrite rules can be applied in any *surroundings* (context), generating the rewrite relation. In the first-order case one speaks of TRSs, while in the higher-order case there are several conceptually similar, but notationally often quite different, proposals. The first general higher order format was introduced long ago by Klop [Klo80] under the name of *Combinatory Reduction Systems* (CRSs). Since then, several other interesting formalisms have been introduced [Kha92, Wol93, Nip93, LS93, VOvR94]. Restricted rewriting systems with substitutions were first studied by Pkhakadze [Pkh77] and Aczel [Acz78]. See van Raamsdonk [VR96] for a detailed comparison of various forms of higher-order rewriting.

It is often of interest to have the possibility of putting restrictions on the generation of either the terms or the rewrite relation or both. For example, many typed lambda calculi (such as the simply typed λ-calculus and the system **F** [P.92]) can be viewed as untyped lambda calculi with restrictions on the formation of *terms*. (See [KvOvR93] for an encoding of the system **F** as a *substructure CRS*.) On the other hand, many rewrite strategies are naturally expressed by restricting the application of the *rewrite rules*. The call-by-value strategy in λ-calculus [Plo75], for example, can be specified by restricting the second *argument* of the β-rule to values. In general, restricting arguments gives rise to so-called *conditional* ERSs (cf. [BK82]). The leftmost-outermost strategy can be specified by restricting the *context* in which the β-rule may be applied. We will call the latter kind of rules in which contexts are restricted *context-sensitive*.[3] We will now introduce *CCERSs* which allow all three kinds of restriction.

2.1.1 The syntax of CCERSs

CCERSs are an extension of ERSs, which are based on the syntax of Pkhakadze [Pkh77]. Terms in CCERSs are built from the alphabet just like they are in the first-order case. The symbols having binding power (like the λ in λ-calculus and the $\int$ in integrals) require some binding variables and terms as arguments, as specified by their *arity*. *Scope indicators* are used to specify which variables have binding power in which arguments. For example, a β-redex in the λ-calculus appears as $Ap(\lambda x\ t, s)$, where Ap is a function symbol of arity 2 and λ is an operator sign of arity $(1,1)$ and scope indicator (1). Integrals such as $\int_s^t f(x)\,dx$ can be represented as $\int x(s,t,f(x))$ by using an operator sign $\int$ of arity $(1,3)$ and scope indicator (3).

Metaterms will be used to write rewrite rules. They are constructed from *metavariables* and meta-expressions for substitutions, called *metasubstitutions*. Instantiation of metavariables in metaterms yields terms. Metavariables play the rôle of variables in the TRS rules and of function variables in other formats of higher-order rewriting such as Higher-Order TRSs (HOTRSs) [Wol93], Higher-Order Rewrite Systems (HRS) [Nip93], and Higher-Order Rewriting Systems (HORSs) [VOvR94]. Unlike the function variables in HOTRSs, HRSs, and HORSs, however, metavariables *cannot* be bound.

Definition 2.2 *Let Σ be an* alphabet *comprising infinitely many* variables*, denoted by x, y, $z,\ldots$, and* symbols *(*signs*). A symbol σ can be either a* function symbol *(*simple operator*) having an* arity *$n \in \mathcal{N}$ or an* operator sign *(*quantifier sign*) having* arity *$(m,n) \in \mathcal{N}^+ \times \mathcal{N}^+$. If it is an operator sign it needs to be supplied with m* binding variables *$x_1,\ldots,x_m$ to form a* quantifier *(*compound operator*) $\sigma x_1 \ldots x_m$, and it also has a* scope indicator *specifying in which of the n arguments it has binding power.*[4] Terms

[3]The distinction between 'conditional' and 'context-sensitive' is, however, more historical than conceptual.

[4]Scope indicators can be avoided at the expense of side conditions of the form $x \notin FV(s)$. In this case, in order to avoid unintended bindings, such conditions must be imposed on construction of (admissible) terms rather than on the usage of rewrite rules.

t, s, e, o are constructed from variables, function symbols, and quantifiers in the usual first-order way, respecting (the second component of the) arities. A predicate AT on terms specifies which terms are admissible.

Metaterms *are constructed like terms, but also allowing* metavariables *A, B,... and* metasubstitutions $(t_1/x_1,\ldots,t_n/x_n)t_0$*, where each t_i is an arbitrary metaterm and the x_i have a binding effect in t_0. Metaterms without metasubstitutions are called* simple. *An* assignment *θ maps each metavariable to a term. The application of θ to a metaterm t is written $t\theta$ and is obtained from t by replacing metavariables with their values under θ and by replacing metasubstitutions $(t_1/x_1,\ldots,t_n/x_n)t_0$, in right to left order, with the result of substitution of terms $t_1,\ldots,t_n$ for free occurrences of $x_1,\ldots,x_n$ in t_0. The substitution operation may involve a* renaming *of bound variables to avoid collision, and we assume that the set of variables in Σ comes equipped with an equivalence relation, called renaming, such that any equivalence class of variables is infinite. We also assume that any variable can be renamed by any other variable in the corresponding equivalence class.*[5] *Unless otherwise specified, the default renaming relation is the total binary relation on variables (a partial renaming relation may be useful for conditional systems).*

The specification of a CCERS consists of an alphabet (generating a set of terms possibly restricted by the predicate AT as specified above), and a set of rules (generating the rewrite relation possibly restricted by *admissibility* predicates AA and AC as specified below). The predicate AT can be used to express sorting and typing constraints, since sets of admissible terms allowed for arguments of an operator can be seen as terms of certain sorts or types. The predicates AA and AC impose restrictions respectively on arguments of (admissible) redexes and on the contexts in which they can be contracted.

The CCERS syntax is very close to the syntax of the λ-calculus. Those already familiar with the λ-calculus may therefore find ERSs easier to understand than CRSs, although the differences between the two are 'semantically' insignificant. See also [VR96]. For example, the β-rule is written as $Ap(\lambda xA, B) \to (B/x)A$, where A and B can be instantiated by any terms. The η-rule is written as $\lambda xAp(A, x) \to A$, where for any assignment $\theta \in AA(\eta)$, $x \notin FV(A\theta)$ (the set of free, i.e., unbound, variables of $A\theta$); otherwise an x occurring free in $A\theta$ and therefore bound in $\lambda xAp(A\theta, x)$ would become free. A rule like $f(A) \to \exists x(A)$ is also allowed, but in that case the assignment θ with $x \in A\theta$ is not allowed. Such a collision between free and bound variables cannot arise when assignments are restricted by the condition $(*)$, described below.

Familiar rules for defining existential quantifier $\exists x$ and the quantifier $\exists!x$ (there exists exactly one x) are written as $\exists x(A) \to (\tau x(A)/x)A$ and $\exists!x(A) \to \exists x(A) \wedge \forall x\forall y(A \wedge (y/x)A \Rightarrow x = y)$, respectively. For the assignment associating $x = 5$ to the metavariable A, these rules generate rewrite steps $\exists x(x = 5) \to \tau x(x = 5) = 5$ and $\exists!x(x = 5) \to \exists x(x = 5) \wedge \forall x\forall y(x = 5 \wedge y = 5) \Rightarrow x = y)$. In general, evaluation of a reduction step may involve execution of a number of substitutions corresponding to the metasubstitutions in the right-hand-side of the rule. This will be explained by examples in the next section.

[5] An equivalence class of variables can, for example, be the set of variables of the same type in a typed language.

Definition 2.3 *A* Context-sensitive Conditional Expression Reduction System *(CCERS) is a pair* (Σ, R)*, where* Σ *is an* alphabet *described in Definition 2.2 and* R *is a set of* rewrite rules $r : t \to s$*, where* t *and* s *are closed metaterms (i.e., metaterms possibly containing 'free' metavariables but not containing free variables).*

Furthermore, each rule r *has a set of* admissible assignments $AA(r)$ *which, to prevent confusion of variable bindings, must satisfy the following condition of being* variable-capture-free*:*

(*) for any assignment $\theta \in AA(r)$*, any metavariable* A *occurring in* t *or* s*, and any variable* $x \in FV(A\theta)$*, either every occurrence of* A *in* r *is in the scope of some binding occurrence of* x *in* r *or no occurrences are.*

For any $\theta \in AA(r)$*,* $t\theta$ *is an* r-redex *or an* R-redex *(and so is any* variant *of* $t\theta$ *obtained by renaming of bound variables), and* $s\theta$ *is the* contractum *of* $t\theta$*. We call* R simple *if the right-hand sides of* R*-rules are simple metaterms. We call redexes that are instances of the same rule* weakly similar*.*

Furthermore, each pair (r, θ) *with* $r \in R$ *and* $\theta \in AA(r)$ *has a set* $AC(r, \theta)$ *of* admissible contexts *such that if a context* $C[\,]$ *is admissible for* (r, θ) *and* o *is the contractum of* $u = r\theta$ *according to* r*, then* $C[u] \to C[o]$ *is an* R*-reduction step. In this case,* u *is* admissible *for* r *in the term* $C[u]$*. We require that the set of admissible terms be closed under reduction. We also require that admissibility of terms, assignments, and contexts be closed under the renaming of bound variables.*[6]

We call a CCERS context-free*, or simply a* Conditional Expression Reduction System *(CERS), if every term is admissible, if every context is admissible for any redex, if the rules* $r : t \to s$ *are such that* t *is a simple metaterm and is not a metavariable, and if each metavariable that occurs in* s *also occurs in* t*. Moreover if for any rule* $r \in R$*,* $AA(r)$ *is the maximal set of variable-capture-free assignments, then we call the CERS an* unconditional *Expression Reduction System, or simply an Expression Reduction System (ERS).*[7]

Note that in CCERSs (but not in CERSs or ERSs) we allow metavariable-rules like $\eta^{-1} : A \to \lambda x Ap(A, x)$ and metavariable-introduction-rules like $f(A) \to g(A, B)$, which are usually excluded a priori. This is useful only when the system is conditional. Like in the η-rule,the requirement $(*)$ forces $x \notin FV(A\theta)$ for every $\theta \in AA(\eta^{-1})$.

Let $r : t \to s$ be a rule in a CCERS R and let θ be admissible for r. Subterms of a redex $v = t\theta$ that correspond to the metavariables in t are the *arguments* of v, and the rest of v is the *pattern* of v (hence the binding variables of the quantifiers occurring in the pattern also belong to the pattern). Subterms of v whose head symbols are in its pattern are called the *pattern-subterms* of v. The pattern of the right-hand side of a simple CCERS rule is defined similarly.

Notation We use a, b, c, d for constants, use t, s, e, o for terms and metaterms, use u, v, w for redexes, and use N, P, Q for reductions (i.e., reduction paths). We write $s \subseteq t$ if s is a subterm (occurrence) of t. A one-step reduction in which a redex $u \subseteq t$ is contracted

[6]Closure of admissibility of contexts under the renaming of bound variables may need some clarification: We mean that if u is admissible in $C[u]$ and if $C'[u']$ is its variant (obtained by a renaming of bound variables in $C[u]$), then u' must be a redex admissible for $C'[\,]$.

[7]The renaming relation for ERSs is total.

is written as $t \xrightarrow{u} s$ or $t \to s$ or just u. We write $P : t \twoheadrightarrow s$ or $t \overset{P}{\twoheadrightarrow} s$ if P denotes a reduction (sequence) from t to s, write $P : t \twoheadrightarrow$ if P may be infinite, and write $P : t \twoheadrightarrow \infty$ if P is infinite (i.e, of the length ω). For finite P, $P + Q$ denotes the concatenation of P and Q.

Below, when we refer to terms and redexes, we will always mean admissible terms and admissible redexes except that are explicitly mentioned.

2.1.2 Expressive power of CCERSs

To avoid a significant deviation from the main theme, how to encode conditional TRSs [BK86] and reduction strategies as CCERSs is described in this subsection only very briefly. More complex examples, namely, encodings of Hilbert- and Gentzen-style proof systems and π-calculus into CCERSs are shown in [KvO95b, KOvO01a]. For more details refer to Khasidashvili and van Oostrom [KvO95b].

Conditional TRSs

Conditional term rewriting systems (CTRSs) were introduced by Bergstra and Klop [BK86]. Their conditional rules have the form $t_1 = s_1 \wedge \cdots \wedge t_n = s_n \Rightarrow t \to s$, where s_i and t_i may contain variables in t and s. According to such a rule, $t\theta$ can be rewritten to $s\theta$ if all the equations $s_i\theta = t_i\theta$ are satisfied. CTRSs were classified depending on how satisfaction is defined ('=' can be interpreted as $\twoheadrightarrow$, $\leftrightarrow^*$, etc.) As Bergstra and Klop remark this can be generalized by allowing for arbitrary predicates on the variables as conditions (cf. also [DOG88, Toy89]).

Clearly, all these CTRSs are context-free CCERSs since they allow conditions on the arguments but not on the context of rewrite rules. For this reason results for them are sometimes a special case of general results holding for all CCERSs. In particular, *stable* CTRSs for which the unconditional version is orthogonal as defined in [BK86] are orthogonal in our sense (to be defined in Subsection 2.2) and so are confluent.

Encoding of strategies

In the literature a strategy for a rewriting system (R, Σ) is often defined as a map $F{:}\,Ter(\Sigma) \to Ter(\Sigma)$, such that $t \to F(t)$ if t is not a normal form, and $t = F(t)$ otherwise (e.g., [Bar84]). Such strategies are deterministic and do not specify the way in which to obtain $F(t)$ from t.

The first thing to take into account here is that in a term there may be disjoint redex occurrences yielding the same result if reduced. For example, take simply the TRS $R = \{f(x) \to a,\ b \to b\}$ and the term $t = g(b, f(b))$. Then t is rewritten to itself when either the first or the second occurrence of b in it is rewritten (using the second rule). The leftmost b is *essential* (i.e., contributes to the normal form) [Kha93], whereas the rightmost b is not. Here our knowing that a strategy F rewrites t to t is not enough to tell us whether F rewrites an essential redex in t or an inessential one. Similarly, $I(Ix)$ can be β-reduced in one step to Ix, where $I = \lambda x.x$, but the information $I(Ix) \to Ix$ is not enough to determine whether the outermost redex has been contracted or the innermost one (the effect that contraction of different redexes yields the same result is called a 'syntactic accident' [Lév78]). So a strategy should specify which redex occurrence must be contracted.

The second thing to take into account is that a redex occurrence can be an instance of more than one rule. That is, $LHS(r_1)\theta_1 = u = LHS(r_2)\theta_2$ for some rules r_1 and r_2 and some assignments $\theta_1 \in AA(r_1)$ and $\theta_2 \in AA(r_2)$. And the contracta of the different redexes can be the same, which shows that even knowing the occurrence of the redex may not be sufficient for knowing which rule has been applied. For example, consider the rules for parallel *or*:

$$or(true, x) \to true, or(x, true) \to true.$$

Then $or(true, true) \to true$ by applying either of the two rules. So a strategy should specify which rule must be applied.

Finally, although for orthogonal ERSs the result of a reduction step from some term t is uniquely determined by the redex occurrence and the rule to be applied, this need not be the case in general. For example, applying the (variable-introducing, hence non-orthogonal) rule $a \to A$ to the term a in the empty context may lead to any result, depending on the assignment to A.

Thus we prefer to view a strategy as a set F of triples $(r, \theta, C[\,])$ specifying that rule $r : t \to s \in R$ can be used with assignment θ in context $C[\,]$ to rewrite $C[t\theta]$ to $C[s\theta]$.[8] Thus a strategy F may be non-deterministic in that the redex to be contracted in a term t can be selected from a possibly non-singleton set of redexes of t specified by F. To a strategy F one can associate a CCERS R_F encoding exactly the same information by taking $\theta, C[\,]$ admissible for r iff $(r, \theta, C[\,]) \in F$. Obviously, this also holds the other way around; that is, every CCERS can be viewed as a strategy for its unconditional version.

Note that the set of terms $und(F)$ on which a strategy F (considered as a set of triples) is undefined need not coincide with the set of normal forms. Indeed, many strategies halt once they reach terms to a set of *values* (e.g., head normal forms or weak head normal forms in the λ-calculus), or if a *deadlock* situation arises; see [Len97b] for a number of such strategies. So our definition provides for such strategies, except the information about which terms from $und(F)$ are values (and which correspond to a deadlock situation) must be added explicitly.

2.2 Orthogonal CCERSs

In this section, we introduce a suitable concept of orthogonality for CCERSs, prove confluence for them, and illustrate how this result can be used for proving confluence for restricted λ-calculi. We then recall some results concerning the *similarity* of redexes [Kha94a] in orthogonal CCERSs. Finally, we present a new proof of the existence of external redexes [HL91] in every reducible term in an orthogonal fully-extended CCERS. The results concerning the similarity of redexes and external redexes will be used later on to study the perpetuality of redexes in orthogonal fully-extended CCERSs.

[8]Note that an ordinary strategy F can be directly encoded by associating the set $\{(r : t \to s, \theta, C[\,]) \mid r \in R, C[s\theta] = F(C[t\theta])\}$ to it.

2.2.1 Orthogonality and confluence

The idea of orthogonality is that contraction of a redex does not destroy other redexes (in whatever way) but instead leaves a number of their residuals. A prerequisite for the definition of residual is the concept of *descendant*, also called *trace*, which allows the tracing of subterms along a reduction. Whereas this concept is pretty simple in the first-order case, CCERSs may exhibit complex behaviour due to the possibility of nested metasubstitutions in the right-hand sides of rules, thereby complicating the definition of descendants. A standard technique in higher-order rewriting [Klo80] (illustrated below on examples) is to *decompose* or *refine* each rewrite step into two parts: a *TRS*-part in which the left-hand side is replaced by the right-hand side without evaluating the (meta)substitutions, and a *substitution*-part in which the delayed substitutions are evaluated. To express substitution we use the S-reduction rules

$$S^{n+1}x_1 \dots x_n A_1 \dots A_n A_0 \to (A_1/x_1, \dots, A_n/x_n)A_0, \quad n = 1, 2, \dots,$$

where S^{n+1} is the *operator sign of substitution* with arity $(n, n+1)$ and scope indicator $(n+1)$ and where $x_1, \dots, x_n$ and $A_1, \dots, A_n, A_0$ are pairwise distinct variables and metavariables. (We assume that the CCERS does not contain symbols S^{n+1}; it can of course contain a renamed variant of S-rules. The collection of all substitution rules, renamed or not, is an ERS itself.) Thus S^{n+1} binds only in the last argument. One can think of S-redexes as (simultaneous) let-expressions.

Thus the descendant relation of a rewrite step can be obtained by composing the descendant relation of the TRS-step and the descendant relations of the S-reduction steps. All known concepts of descendants agree in the cases when the subterm $s \subseteq t$ which is to be traced during a step $t \overset{u}{\to} o$ is (1) in an argument of the contracted redex u, (2) properly contains u, or (3) does not overlap with u. The concepts differ when s is a pattern-subterm (i.e., when s is in the contracted redex u but is not in any of its arguments), in which case we define the contractum of u to be the descendant of s. According to many definitions, however, s does not have a u-descendant (*descendant* is often used as a synonym of *residual*, which it is not). In the case of TRSs, our definition coincides with Boudol's [Bou85] and differs slightly from Klop's [Klo92]: according to Klop's definition the descendants of a contracted redex, as well as of any of its pattern-subterms, are all subterms whose head-symbols are within the pattern of the contractum.

We first explain our descendant concept by using examples. Consider a TRS-step $t = f(g(a)) \to h(b) = s$ performed according to the rule $f(g(x)) \to h(b)$. The descendant of both pattern-subterms $f(g(a))$ and $g(a)$ of t in s is $h(b)$[9] and a does not have a descendant in s. The refinement of a β-step $t = Ap(\lambda x(Ap(x,x)), z) \to_\beta Ap(z,z) = e$ would be $t = Ap(\lambda x(Ap(x,x)), z) \to_{\beta_f} o = S^2 xz Ap(x,x) \to_S Ap(z,z) = e$: the descendant of both t and $\lambda x(Ap(x,x))$ after the *TRS*-step is the contractum $S^2 xz Ap(x,x)$, the descendants of $Ap(x,x), z \subseteq t$ are the respective subterms $Ap(x,x), z \subseteq o$, the descendant of both $o = S^2 xz Ap(x,x)$ and $Ap(x,x)$ after the substitution step is the contractum e, and the descendants of $z \subseteq o$, as well as of the bound occurrence of x in $Ap(x,x)$, are the occurrences of z in e.

[9]According to Klop's definition, the occurrence of b in $h(b)$ is also a descendant for both $f(g(a))$ and $g(a)$.

This definition by example can be formalized using *paths* to refer to subterm positions in a term t. Paths, denoted by ϕ, ψ, ζ, ξ, are strings of integers: the empty string ε refers to the top-position (i.e., the term t itself) and if a path $i_1, \dots, i_k$ refers to a subterm $\sigma x_1 \dots x_m(t_1, \dots, t_n)$ of t, then $i_1, \dots, i_k, i_{k+1}$ is again a path for each $1 \leq i_{k+1} \leq n$ which refers to the subterm $t_{i_{k+1}}$ of t; $\preceq$ denotes the prefix ordering on paths. (The binding variables in a quantifier are considered to be at the same position as the quantifier symbol itself. They therefore can be ignored because they are not subterms.)

Definition 2.4 *Let t be a term in a simple CCERS R (so the refinement of an R-step coincides with the R-step itself), let $r : t' \to s' \in R$, let u be an (admissible) r-redex in t occurring at a position ϕ, let $t \overset{u}{\to} s$, and let o be a subterm of t at a position ψ.*

1. *If ϕ and ψ are disjoint (i.e, neither $\phi \preceq \psi$ nor $\psi \preceq \phi$), then the descendant of o is the subterm of s at the same position ψ;*

2. *If $\psi \preceq \phi$, then again the descendant of o is the subterm of s at the same position ψ;*

3. *If $\psi = \phi \cdot \zeta$ where ζ is a nonvariable position in the left-hand-side t' of r ($\cdot$ is the concatenation operation on paths), then the descendant of o is the subterm of s at the position ϕ (i.e., is the contractum of u);*

4. *If $\psi = \phi \cdot \zeta_i \cdot \xi$ where ζ_i is the position of the ith-from-the-left variable occurrence in t', then the descendants (0 or more) of o are the subterms in s at all positions $\psi_j = \phi \cdot \zeta^i_j \cdot \xi$, $1 \leq j \leq k_i$, where $\zeta^i_1, \dots, \zeta^i_{k_i}$ are the positions of all occurrences of the same variable in the right-hand-side s' of r.*

Definition 2.5 *Let $S^{n+1}x_1 \dots x_n t_1 \dots t_n t_0$ be an S-redex in a term t at a position ϕ in a CCERS, let $t \overset{u}{\to}_S s$, and let o be a subterm of t at a position ψ.*

1. *If ϕ and ψ are disjoint, then the descendant of o is the subterm of s at the same position ψ;*

2. *If $\psi \preceq \phi$, then again the descendant of o is the subterm of s at the same position ψ;*

3. *If $\psi = \phi \cdot n+1 \cdot \xi$ (i.e., $o \subseteq t_0$), then the descendant of o is the subterm in s at position $\phi \cdot \xi$.*

4. *If $\psi = \phi \cdot i \cdot \xi$ where $1 \leq i \leq n$, then the descendants (0 or more) of o are the subterms in s at all positions $\psi_j = \phi \cdot \zeta^i_j \cdot \xi$, $1 \leq j \leq k_i$, where $\zeta^i_1, \dots, \zeta^i_{k_i}$ are the positions of all occurrences of x_i in t_0.*

To illustrate further the third and the fourth cases of Definition 2.5, consider the S-reduction step $t = Sxf(a)g(x) \to_S g(f(a)) = s$. Then the descendant of $x \subseteq t$ is $f(a) \subseteq s$, and the descendant of $g(x) \subseteq t$ is s. The descendants of $f(a), a \subseteq t$ are the occurrences $f(a), a \subseteq s$, respectively.

The *descendant* concept extends by transitivity to arbitrary reductions consisting of TRS-steps and S-reduction steps. If P is an R-reduction, then *P-descendants* are defined to be the descendants under the refinement of P. The *ancestor* relation is the inverse of the descendant relation. The descendant concept allows us to define residuals:

Definition 2.6 *Let $t \xrightarrow{u} s$ be in a CCERS R, let $v \subseteq t$ be an admissible redex, and let $w \in s$ be a u-descendant of v. We call w a u-*residual *of v if (a) the patterns of u and v do not overlap (i.e., the pattern-occurrences do not share an occurrence of a symbol in t), (b) w is a redex weakly similar to v (see Definition 2.3), and (c) w is admissible. (So u itself does not have u-*residuals *in s.) The concept of a* residual *of redexes extends naturally to arbitrary reductions. A redex in s is called a* new *redex or a* created *redex if it is not a residual of a redex in t. The* predecessor *relation is inverse to that of residual.*

Definition 2.7 *We call a CCERS* orthogonal *if:*

- *the left-hand sides of rules are not single metavariables,*
- *the left-hand side of any rule is a simple metaterm and its metavariables contain those of the right-hand side, and*
- *all the descendants of an admissible redex u in a term t under the contraction of any other admissible redex $v \subseteq t$ are residuals of u.*

The second condition ensures that rules exhibit deterministic behaviour when they can be applied. The last condition is the counterpart of the *subject reduction property* in typed λ-calculi [P.92]. For example, consider the rules $a \to b$ and $f(A) \to A$ with the admissible assignment $A\theta = a$. The descendant $f(b)$ of the redex $f(a)$ after contraction of a is not a redex because the assignment $A\theta = b$ is not admissible. Hence the system is not orthogonal.

Definition 2.8 *Reductions starting from the same term are called* co-initial. *Recall that co-initial reductions $P : t \twoheadrightarrow s$ and $Q : t \twoheadrightarrow e$ are* weakly equivalent *or* Hindley-equivalent *[Bar84], written $P \approx_H Q$, if $s = e$ and the residuals of any redex of t under P and under Q are the same redexes in s. Furthermore, P and Q are* strictly equivalent *[Kha90], written $P \approx_{st} Q$, if $s = e$ and the descendants of any subterm of t under P and under Q are the same subterms in s.*

Using these equivalences and the above definition of residuals, we can easily infer *strong* [Lév78, HL91] and *strict* [Kha90] forms of the Church-Rosser property for CCERSs.

A standard method of proving the *strong* version of CR is one using FD and the fact that any pair of redexes u, v in a term *strongly commute*: $u + v/u \approx_H v + u/v$ [Lév78]; that latter property will be called *strong local confluence.*[10] Indeed, as in orthogonal TRSs [HL91], the λ-calculus [Lév80, Bar84], orthogonal CRSs [Klo80], and orthogonal

[10]FD is often referred to the stronger property that all developments of a set of redexes in a term are terminating and all complete developments of the same set of redexes are Hindley-equivalent. This stronger version follows easily from the weaker version (i.e., termination of all developments) and the strong commutativity of co-initial steps.

HRSs [VO94], one can in orthogonal CCERSs use FD and strong commutativity to define for any co-initial reductions P and Q the *residual of P under Q*, written P/Q. We write $P \trianglelefteq_L Q$ if $P/Q = \emptyset$ ($\trianglelefteq_L$ is the *Lévy-embedding* relation); P and Q are called *Lévy-equivalent* or *permutation-equivalent* (written $P \approx_L Q$) if $P \trianglelefteq_L Q$ and $Q \trianglelefteq_L P$. It follows from the definition of / that if $P + P'$ and $Q + Q'$ are co-initial finite reductions in an orthogonal CCERS, then $(P+P')/Q \approx_L P/Q + P'/(Q/P)$ and $P/(Q+Q') \approx_L (P/Q)/Q'$. This is all well known and we do not give more details. The strong Church-Rosser theorem then states that, for any co-initial finite reductions P and Q in an orthogonal ERS, $P \sqcup Q \approx_L Q \sqcup P$, where $P \sqcup Q$ means $P + Q/P$. The *Strict Church-Rosser* theorem states that, for any co-initial finite reductions P and Q in an orthogonal ERS, $P \sqcup Q \approx_{st} Q \sqcup P$. (Thus, $P \approx_L Q$ implies $P \approx_{st} Q$.) Like the strong CR property, the strict CR property follows from FD and the following *strict local confluence* property: any two co-initial steps u, v *strictly commute*: $u \sqcup v \approx_{st} v \sqcup u$.

Since developments in CCERSs are obtained by restricting developments in ERSs, and the latter are a special case of developments in PRSs [VR96] which are finite [VO97], we obtain the following result.

Theorem 2.9 (Finite Developments) *All developments of a term t in an orthogonal CCERS R eventually terminate.*

Using this theorem and the last condition in the definition of orthogonality, the next theorem follows from some abstract theory of residuals.

Theorem 2.10 *Let P and Q be any co-initial finite reductions in an orthogonal CCERS R. Then*

(1) **(Strong Church-Rosser)** $P \sqcup Q \approx_L Q \sqcup P$.

(2) **(Strict Church-Rosser)** $P \sqcup Q \approx_{st} Q \sqcup P$.

The λ-calculus [Bar84] is the prime example of an ERS. If one restricts term formation in it, one arrives at a large class of typed lambda calculi. Since the rewrite relation in these calculi is not restricted in any way and typed terms are closed under β-reduction,[11] these CCERSs are orthogonal, hence confluent. Another example of an orthognal CCERS is the call-by-need λ-calculus of Ariola et al. [AFM⁺95], hence confluent [KOvO01a].

An emerging class of context-sensitive conditional ERSs is the class of λ-calculi with restricted expansion rules like $\bar{\eta}$ (see e.g. [Aka93]). These calculi are not orthogonal, but their confluence can be shown by modifying the confluence diagrams arising from FD for the corresponding unconditional expansion rules.

2.2.2 Similarity of redexes

The idea of *similarity* of redexes [Kha94b, Kha94a] u and v is that u and v are weakly similar – that is, they match the same rewrite rule – and quantifiers in the pattern of u and v bind 'similarly' in the corresponding arguments. For example, recall that a

[11]Proving this *subject reduction* property is sometimes nontrivial.

β-redex $Ap(\lambda xt, s)$ is an *I-redex* if $x \in FV(t)$ and is a *K-redex* otherwise. Then all I-redexes are similar and all K-redexes are similar, but no I-redex is similar to a K-redex. Consequently, for any pair of corresponding arguments of u and v, either both are erased after contraction of u and v or none is.

A redex in a CCERS has the form $u = C[t_1, \ldots, t_n]$, where C is the pattern and $t_1, \ldots, t_n$ are the arguments. Sometimes we will write u as $u = C[\overline{x_1}t_1, \ldots, \overline{x_n}t_n]$, where $\overline{x_i} = \{x_{i_1}, \ldots, x_{i_{n_i}}\}$ is the subset of binding variables of C such that t_i is in the scope of an occurrence of each x_{i_j}, $j = 1, \ldots, n_i$. Let us call the maximal subsequence $j_1, \ldots, j_k$ of $1, \ldots, n$ such that $t_{j_1}, \ldots, t_{j_k}$ have u-descendants the *main sequence* of u (or the *u-main sequence*), call $t_{j_1}, \ldots, t_{j_k}$ the (u-)*main arguments*, and call the remaining arguments (u)-*erased*. Further, call u *erasing* if $k < n$ and *non-erasing* otherwise.

Now the similarity of redexes can be defined as follows: weakly similar redexes $u = C[\overline{x_1}t_1, \ldots, \overline{x_n}t_n]$ and $v = C[\overline{x_1}s_1, \ldots, \overline{x_n}s_n]$ are *similar* if, for any $1 \leq i \leq n$, $\overline{x_i} \cap FV(t_i) = \overline{x_i} \cap FV(s_i)$. For example, consider the rule $\sigma x(A, B) \to (\sigma x(f(A), A)/x)B$. Then the redexes $u = \sigma x(x, y)$ and $v = \sigma x(f(x), y)$ are similar, while $w = \sigma x(y, y)$ is not similar to any of them since $x \notin FV(y)$. However, note that the second arguments of all the redexes u, v and w are main and the first arguments are erased. In this chapter, it is more convenient to use a slightly relaxed concept of similarity, written $\sim$, such that $u \sim v \sim w$:

Definition 2.11 *We write $u \sim v$ if the main sequences of u and v coincide and for any main argument t_i of u, $\overline{x_i} \cap FV(t_i) = \overline{x_i} \cap FV(s_i)$.*

The following lemma implies in particular that, indeed, if u and v are similar, then $u \sim v$, and that $\sim$ is an equivalence relation. Because its proof involves properties of essentiality not needed elsewhere in this chapter, we omit the proof and instead refer to previous work [Kha94a]. The lemma is quite intuitive anyway: it shows that only pattern-bindings (i.e., bindings from inside the pattern) of free variables in *main* arguments of a redex are relevant for the erasure of its arguments.

Below, θ will not only denote assignments but will also denote substitutions assigning terms to variables; when we write $o' = o\theta$ for a substitution θ, we assume that no free variables of the substituted subterms become bound in o' (i.e., we rename bound variables in o when necessary).

Lemma 2.12 *Let $u = C[\overline{x_1}t_1, \ldots, \overline{x_n}t_n]$ and $v = C[\overline{x_1}s_1, \ldots, \overline{x_n}s_n]$ be weakly similar redexes, and let for any main argument s_i of v, $\overline{x_i} \cap FV(t_i) \subseteq \overline{x_i} \cap FV(s_i)$. Then the main sequence of u is a subset of the main sequence of v.*

Corollary 2.13 *Let $u = C[\overline{x_1}t_1, \ldots, \overline{x_n}t_n]$ and $v = C[\overline{x_1}s_1, \ldots, \overline{x_n}s_n]$ be weakly similar redexes, and let for any main argument s_i of v, $\overline{x_i} \cap FV(t_i) = \overline{x_i} \cap FV(s_i)$. Then $u \sim v$. In particular, if $u = v\theta$, then $u \sim v$.*

2.2.3 External redexes

In this subsection we will show that every reducible term in an orthogonal *fully-extended* (see Definition 2.14) CCERS has an *external* redex. External redexes for orthogonal TRSs were introduced by Huet and Lévy [HL91], who also proved the existence of external redexes in every reducible term. Both the original definition of external redexes and the existence proof are quite lengthy.

With our concept of descendant, external redexes can be defined as redexes whose descendants can never occur inside the arguments of other redexes. Any external redex is trivially outermost, but an outermost redex is not necessarily external. Contracting a redex *disjoint from it*, may cause its residual to be non-outermost. For example, consider the orthogonal TRS $\{f(x,b) \to c, a \to b\}$. The first a in $f(a,a)$ is outermost but not external; contracting the second a (which is disjoint from it) creates the redex $f(a,b)$ having the residual of the first a as argument. The second a is external.

In an ERS, there may be another reason why an outermost redex need not be external. Contracting a redex *in one of its argument*, may cause its residual to be non-outermost. This already shows up in the $\lambda\beta\eta$-calculus. Let $I = \lambda x.x$ and $K = \lambda xy.x$, as usual [Bar84]. The redex $u = I(KIx)$ in $\lambda x.I(KIx)x$ is outermost but not external; contracting the redex KIx in its argument creates the η-redex $\lambda x.IIx$ having the residual II of u as argument. This example can be readily encoded as an orthogonal ERS. We will see later that because of rules like η which test for the absence of variables in subterms (occur check!) even the conservation theorem fails for orthogonal CCERSs in general. To rule out such rules, following [VO96, HP96], we introduce the concept of *full extendedness* for CCERSs:

Definition 2.14 *We call a CCERS R* fully-extended *iff for any step $t \xrightarrow{u} s$ in R and any occurrence $w \subseteq t$ of an instance of the left-hand-side (of a rule $r \in R$) such that:*

(a) the patterns of w and u in t do not overlap, and

(b) w has a u-descendant $w' \in s$ that is a redex,

w is an admissible redex in t weakly similar to w'.

Now we can easily generalize the proof of existence of external redexes in [Kha93] from orthogonal TRSs to fully-extended orthogonal CCERSs.

Definition 2.15 *Let $P : t \twoheadrightarrow o$ in an orthogonal fully-extended CCERS. A subterm $s \subseteq t$ is P-*external *if no descendants of s along P appear inside redex-arguments and is P-*internal *otherwise. A subterm $s \subseteq t$ is* external *if s is Q-*external *for any finite reduction $Q : t \twoheadrightarrow$; otherwise s is* internal.[12]

Consider the λ-term $t = \Omega((\lambda xy.xy)I(Ix))$, where I is as defined above and $\Omega = (\lambda x.xx)(\lambda x.xx)$, and consider the β-reduction $P : t \xrightarrow{v} \Omega((\lambda y.Iy)(Ix)) \xrightarrow{w} \Omega(I(Ix)) = s$ contracting the redexes $v = (\lambda xy.xy)I$ and $w = (\lambda y.Iy)(Ix)$. Then the redexes $\Omega, v \subseteq t$ are P-external, whereas the redex $Ix \subseteq t$ is P-internal (since after the step v the residual of $Ix \subseteq t$ is inside an argument the created redex w). Note that for the outermost redexes $\Omega, I(Ix) \subseteq s$, there are P-external redexes $\Omega, v \subseteq t$ such that the unique P-descendant of $\Omega \subseteq t$ overlaps the pattern of $\Omega \subseteq s$ and the unique P-descendant of v overlaps the

[12]In [Kha93], an external (resp. P-external) redex is called *unabsorbed* (P*-unabsorbed*).

pattern of $I(Ix) \subseteq s$. Note also that $Ix \subseteq t$ may be $Q \sqcup P$-internal even if it is Q-external. For instance, consider a reduction Q which contracts the occurrences of Ω a finite number of times. These intuitions are formalized in the following three lemmas and are then used to prove the existence of external redexes in reducible terms.

Lemma 2.16 *Let $P : t_0 \xrightarrow{u_0} t_1 \xrightarrow{u_1} \ldots \xrightarrow{u_{n-1}} t_n$ in an orthogonal fully-extended CCERS. Then for any outermost redex $v \subseteq t_n$ there is a P-external redex $u \subseteq t_0$ whose unique P-descendant $s \subseteq t_n$ overlaps the pattern of v (i.e., either $v \subseteq s$ or $s = e$ for some proper pattern-subterm e of v.)*

Proof By induction on $|P|$. If $|P| = 0$ the result is obvious. Suppose $|P| > 0$ and let $P = P' + u_{n-1}$.

(a) Assume first that v is a residual of a redex $v' \subseteq t_{n-1}$. Let $v^* = v'$ if $v' \not\subseteq u_{n-1}$ and let $v^* = u_{n-1}$ otherwise. By full extendedness, since v is outermost, v^* is outermost. By the induction hypothesis there is a P'-external redex $u \subseteq t_0$ whose unique P'-descendant $s' \subseteq t_{n-1}$ satisfies either $v^* \subseteq s'$ or $s' = e'$ for some proper pattern-subterm e' of v^*. Since u is P'-external, s' has a unique descendant s in t_n. If $v^* \subseteq s'$ it is easy to see $v \subseteq s$. Otherwise $e' = s'$ and we consider two cases:

1. $v^* = v'$. Since the patterns of the redexes v' and u_{n-1} do not overlap (by orthogonality), s is a pattern-subterm of v.

2. $v^* = u_{n-1}$. Since the descendant of each pattern-subterm of u_{n-1} is the contractum of u_{n-1}, $v \subseteq s$.

Therefore u is P-external.

(b) Assume now that u_{n-1} creates v. By full extendedness, the contractum of u_{n-1} overlaps the pattern of v. Since v is outermost, u_{n-1} is outermost. By the induction hypothesis there is a P'-external redex $u \subseteq t_0$ such that its unique descendant $s' \subseteq t_{n-1}$ satisfies either $u_{n-1} \subseteq s'$ or $e' = s'$ for some proper pattern-subterm e' of u_{n-1}. Since u is P'-external, s' has a unique descendant s in t_n. Since the descendant of each pattern-subterm of u_{n-1} is the contractum of u_{n-1}, s contains the contractum of u_{n-1}. Thus s overlaps the pattern of v. Therefore u is P-external.

Lemma 2.17 *Let $P : t \twoheadrightarrow s$ be in an orthogonal fully-extended CCERS. If t is reducible, there is a P-external redex u in t.*

Proof If $|P| = 0$ or $|P| > 0$ and s is not a normal form, then the lemma follows immediately from Lemma 2.16. Otherwise, let $P : t \overset{P'}{\twoheadrightarrow} s' \xrightarrow{v} s$. Since s is a normal form, v is outermost. By Lemma 2.16 there is a P'-external redex $u \subseteq t$ whose unique descendant in s' overlaps the pattern of v. Since s has no redexes, u is P-external.

Lemma 2.18 *Let $P : t \twoheadrightarrow s$ and $Q : t \twoheadrightarrow e$ be in an orthogonal fully-extended CCERS. If u is P-internal, then it is $Q \sqcup P$-internal.*

Proof By induction on $|Q|$. It is enough to consider the case when $|Q| = 1$; the rest follows from the induction hypothesis. So let $Q = w$ for a redex w in t. Furthermore, let $P = P^* + v^*$. Without loss of generality we can assume that u is P^*-external, so v^* creates a redex v that contains the unique P-descendant o of u in its argument.

(a) Assume first that o does not have a w/P-descendant. By Theorem 2.10 u does not have $w \sqcup P$-descendants. Hence u is $w \sqcup P$-internal (otherwise its descendants cannot be erased).

(b) Assume now that o has a w/P-descendant o'. Since w/P contracts only residuals of w and v is a new redex, v has a residual v' that contains o' in its argument. By Theorem 2.10 o' is also a $w \sqcup P$-descendant of u. Hence u is $w \sqcup P$-internal.

Theorem 2.19 *Every reducible term in an orthogonal fully-extended CCERS has an external redex.*

Proof Assume that for any outermost redex $u_i \subseteq t$ there is a finite reduction P_i such that u_i is P_i-internal ($i = 1, \ldots, k$). Then by Lemma 2.18 all redexes u_i are P-internal for $P = P_1 \sqcup \ldots \sqcup P_k$. But this is impossible by Lemma 2.17.

2.3 A minimal perpetual strategy

2.3.1 A minimal perpetual strategy

In this section we introduce a perpetual strategy F_m^∞ for orthogonal fully-extended CCERSs by generalizing the *constricting* perpetual strategies in the literature [Pla93, Sør98, Gra96, Mel96, VRSSX99]. We also study properties of F_m^∞ that are used in the next section to obtain new criteria for the perpetuality of redexes and of redex occurrences in orthogonal fully-extended CCERSs. A recent survey on perpetual reductions in the λ-calculus and its extensions can be found in [Sør97a, VRSSX99].

For convenience we have collected the definitions of all related perpetual strategies in Section 2.3.2. To unify the notation we follow [Sør97a, VRSSX99] and use F_1 and F_3 to denote the perpetual strategies of Bergstra and Klop [BK82] and Sørensen [Sør98], respectively. And we use F_z to denote the zoom-in strategy of Melliès [Mel96].

Let us first fix the terminology. Recall that a term t is called *weakly normalizing* (WN), written $WN(t)$, if it is reducible to a *normal form* (i.e., a term without a redex), and t is called *strongly normalizing* (SN), written $SN(t)$, if it does not possess an infinite reduction. We call t an ∞*-term* (written ∞t), if $\neg SN(t)$. Clearly, for any term t, $SN(t) \Rightarrow WN(t)$. If the converse is also true, then we call t *uniformly normalizing* (UN). So a UN term t either does not have a normal form or all reductions from t eventually terminate. Correspondingly, a rewrite system R is called WN, SN, or UN if all terms in R are WN, SN, or UN, respectively.

Following [BK82, Klo92], we call a rewrite step $t \xrightarrow{u} s$, as well as the redex-occurrence $u \subseteq t$, *perpetual* if $\infty t \Rightarrow \infty s$. Otherwise we call them *critical*. We call a redex (not an occurrence) *perpetual* iff its occurrence in every (admissible) context is perpetual. A *perpetual strategy* in an orthogonal fully-extended CCERS is a (partial) function on terms which in any reducible term selects a perpetual redex-occurrence; the orthogonality of the

CCERS implies that the redex-occurrence uniquely determines the rewrite rule (and the corresponding admissible assignment) according to which the redex is to be contracted.

Definition 2.20 *Let $P : t \twoheadrightarrow$ and $s \subseteq t$. Reduction P is* internal *to s if it contracts redexes only in (the descendant of) s. (The contracted redexes in P need not be* proper *subterms of s.)*

Definition 2.21 *(1) Let t be an ∞-term in an orthogonal fully-extended CCERS and let $s \subseteq t$ be a smallest subterm of t such that $\infty(s)$ (i.e., such that every proper subterm $e \subset s$ is SN). Then we call s a* minimal perpetual subterm *of t, and call any external redex of s (such a redex exists by Theorem 2.19) a* minimal perpetual redex *of t.*

(2) Let F^{∞}_{m} be a one-step strategy that contracts a minimal perpetual redex in t if ∞t and otherwise contracts any redex. Then we call F^{∞}_{m} a minimal perpetual strategy*. We call F^{∞}_{m}* constricting *if for any F^{∞}_{m}-reduction $P : t_0 \xrightarrow{u_0} t_1 \xrightarrow{u_1} \dots$ (i.e., any reduction constructed using F^{∞}_{m}) starting from an ∞-term t_0 and for any i, $P^*_i : t_i \xrightarrow{u_i} t_{i+1} \xrightarrow{u_{i+1}} \dots$ is internal to s_i, where $s_i \subseteq t_i$ is the minimal perpetual subterm containing u_i. Constricting minimal perpetual strategies will be denoted F^{∞}_{cm}.*

Recall that, according to Gramlich [Gra96, Remark 3.3.7], a reduction in a TRS is called constricting if it has the form

$$C_0[s_0] \xrightarrow{u_0} C_0[C_1[s_1]] \xrightarrow{u_1} C_0[C_1[C_2[s_2]]] \xrightarrow{u_2} \dots$$

where s_i are minimal perpetual subterms and $u_i \subseteq s_i$. Hence any F^{∞}_{cm}-reduction is constricting (according to Gramlich). Plaisted [Pla93] constructs a constricting perpetual strategy (for TRSs) that in each step contracts a perpetual redex of the leftmost (innermost) minimal perpetual subterm.[13] Sørensen's β-reduction strategy F_3 [Sør98, VRSSX99], as well as Melliès' *zoom-in* β-strategy F_z, produce constricting reductions (on ∞-terms) and are special cases of F^{∞}_{m}. Specifically, F_z is obtained from F^{∞}_{cm} if in each step the leftmost redex of a minimal perpetual subterm is contracted (the leftmost redex in any λ-term is external); and F_3 is a special case of F_z. The perpetual strategy F_2 [VRSSX99] is not zoom-in but is constricting. Note that F^{∞}_{m} is not in general a computable strategy, since SN is already undecidable in orthogonal TRSs [Klo92]; the strategies F_1, F_2, F_3, and F_z are not computable either. These four strategies all produce standard reductions.

Lemma 2.22 *Let t be an ∞-term in an orthogonal fully-extended CCERS, let $s \subseteq t$ be a minimal perpetual subterm of t, and let $P : t \twoheadrightarrow \infty$ be internal to s. Then exactly one residual of any external redex u of s is contracted in P.*

Proof Let $t = C[s]$ and $s = C'[s_1, \dots, u, \dots, s_n]$, where C' consists of the symbols on the path from the top of s to u (the context C' can be empty, in which case $s = u$). If, on the contrary, P does not contract a residual of u, then every step of P takes place either in one of the s_i or in the arguments of u (since u is external in s). Hence at least one of

[13]As noted by Gramlich [Gra96], Plaisted's original definition of 'constricting' is not correct because any infinite reduction becomes constricting.

these subterms has an infinite reduction – a contradiction, since s is a minimal perpetual subterm. Since u is external, P cannot duplicate its residuals; hence P contracts exactly one residual of u.

The following theorem justifies the terminology 'minimal perpetual redex'.

Theorem 2.23 *F_m^∞ is a perpetual strategy in any orthogonal fully-extended CCERS.*

Proof Suppose $\infty(t_0)$, let s_0 be a minimal perpetual subterm of t_0, and let $u \subseteq s_0$ be a minimal perpetual redex. Let $P : t_0 \xrightarrow{u_0} t_1 \xrightarrow{u_1} t_2 \twoheadrightarrow \infty$ be internal to s_0. By Lemma 2.22 exactly one residual of u, say u_i, is contracted in P. Let $P_{i+1} : t_0 \xrightarrow{u_0} t_1 \xrightarrow{u_1} \dots \xrightarrow{u_i} t_{i+1}$ and $P^*_{i+1} : t_{i+1} \xrightarrow{u_{i+1}} t_{i+2} \twoheadrightarrow \infty$ (i.e., $P : t_0 \overset{P_i}{\twoheadrightarrow} t_i \xrightarrow{u_i} t_{i+1} \overset{P^*_{i+1}}{\twoheadrightarrow}$). Since P_i and u are co-initial, $u + P_i/u \approx_L P_i + u/P_i = P_i + u_i = P_{i+1}$ by Theorem 2.10, hence $P = P_{i+1} + P^*_{i+1} \approx_L u + P_i/u + P^*_{i+1}$. That is, u is a perpetual redex-occurrence. Hence F_m^∞ is perpetual.

Definition 2.24 *F_m^∞ is the* leftmost *minimal perpetual strategy, denoted F_{lm}^∞, if in each term it contracts the leftmost minimal perpetual redex. (See Definition 2.31 for the definition of F_{lm}^∞ for the case of the λ-calculus.)*

Theorem 2.25 *F_{lm}^∞ is a constricting strategy in any orthogonal fully-extended CCERS.*

Proof Let $P : t_0 \xrightarrow{u_0} t_1 \xrightarrow{u_1} t_2 \twoheadrightarrow \infty$ be a leftmost minimal perpetual reduction, and let $s_i \subseteq t_i$ be the leftmost minimal perpetual subterm of t_i. Since by Theorem 2.23 u_i is perpetual for the term s_i, the descendant of s_i is an ∞-term and thus contains s_{i+1}, and it is immediate that P is constricting.

Although we do not use it in the following, it is interesting to note that the constricting perpetual reductions are minimal w.r.t. Lévy's embedding relation $\trianglelefteq_L$. Hence the term *minimal.*

The relations $\trianglelefteq_L, \approx_L$, and $/$ (defined in Section 2.2) are extended to possibly infinite co-initial reductions N, N' as follows. $N \trianglelefteq_L N'$, or equivalently, $N/N' = \emptyset$ if for any redex v contracted in N, say $N = N_1 + v + N_2$, $v/(N'/N_1) = \emptyset$ (see the diagram below); and $N \approx_L N'$ iff $N \trianglelefteq_L N'$ and $N' \trianglelefteq_L N$. Here, for any infinite P, $u/P = \emptyset$ if $u/P' = \emptyset$ for some finite initial part P' of P. And P/Q is defined only for finite Q as the reduction whose initial parts are residuals of initial parts of P under Q.

$$\begin{tikzcd} {} \arrow[r, two heads, "N_1"] \arrow[d, two heads, "N'"'] & {} \arrow[r, "v"] \arrow[d, two heads, "N'/N_1"] & {} \arrow[r, two heads, "N_2"] & N \\ {} & {} \end{tikzcd}$$

Theorem 2.26 *Let $P : t_0 \xrightarrow{u_0} t_1 \xrightarrow{u_1} t_2 \twoheadrightarrow \infty$ be a constricting minimal perpetual reduction in an orthogonal fully-extended CCERS and let $Q : t_0 \twoheadrightarrow \infty$ be any infinite reduction such that $Q \trianglelefteq_L P$. Then $Q \approx_L P$.*

Proof Since P is constricting, there is a minimal perpetual subterm $s_0 \subseteq t_0$ such that P is internal to s_0. Since $Q \trianglelefteq_L P$, Q is internal to s_0 as well. By the construction, u_0 is an external redex in s_0, and by Lemma 2.22 exactly one residual u' of u_0 is contracted in Q. So let $Q : t_0 \overset{Q_j}{\twoheadrightarrow} t'_j \xrightarrow{u'} t'_{j+1} \overset{Q^*_{j+1}}{\twoheadrightarrow} \infty$. Then $Q \approx_L u_0 + Q_j/u_0 + Q^*_{j+1}$, and obviously $u_0 \trianglelefteq_L Q$. Similarly, since P is constricting, for any finite initial part P' of P, $P' \trianglelefteq_L Q$, and therefore $P \trianglelefteq_L Q$. Thus $Q \approx_L P$.

2.3.2 Known perpetual strategies of λ-calculus

In this section we collect definitions of all perpetual strategies mentioned in the previous section.

Perpetual strategies on λ-terms will be defined by induction on the structure of terms not in β-normal form, and the redex chosen by a strategy for contraction will be indicated here by underlining. SN_β (resp. NF_β) will denote the set of strongly β-normalizing λ-terms (resp. the set of λ-terms in β-normal form). $\bar{t}$ will denote a sequence of λ-terms $t_1, \cdots, t_n$ and $\bar{t} \in S$ will denote $t_i \in S$ for each i.

Definition 2.27 *([BBKH76]) The β-reduction strategy F_∞ is defined as follows:*

$$\begin{array}{lcll}
F_\infty(x\bar{t}s\bar{o}) & = & x\bar{t}F_\infty(s)\bar{o} & \textit{if } \bar{t} \in NF_\beta, s \notin NF_\beta \\
F_\infty(\lambda x.t) & = & \lambda x.F_\infty(t) & \\
F_\infty((\lambda x.t)s\bar{o}) & = & (\lambda x.t)F_\infty(s)\bar{o} & \textit{if } x \notin FV(t),\ s \notin NF_\beta \\
F_\infty((\lambda x.t)s\bar{o}) & = & \underline{(\lambda x.t)s}\bar{o} & \textit{if } x \in FV(t) \textit{ or } s \in NF_\beta
\end{array}$$

Definition 2.28 *([BK82]) The β-reduction strategy F_1 (called F by Bergstra and Klop [BK82]) is defined as follows:*

$$\begin{array}{lcll}
F_1(x\bar{t}s\bar{o}) & = & x\bar{t}F_1(s)\bar{o} & \textit{if } \bar{t} \in NF_\beta, s \notin NF_\beta \\
F_1(\lambda x.t) & = & \lambda x.F_1(t) & \\
F_1((\lambda x.t)s\bar{o}) & = & (\lambda x.t)F_1(s)\bar{o} & \textit{if } s \notin SN_\beta \\
F_1((\lambda x.t)s\bar{o}) & = & \underline{(\lambda x.t)s}\bar{o} & \textit{if } s \in SN_\beta
\end{array}$$

Definition 2.29 *([VRSSX99]) The β-reduction strategy F_2 is defined as follows:*

$$\begin{array}{lcll}
F_2(x\bar{t}s\bar{o}) & = & x\bar{t}F_2(s)\bar{o} & \textit{if } \bar{t} \in SN_\beta, s \notin SN_\beta \\
F_2(\lambda x.t) & = & \lambda x.F_2(t) & \\
F_2((\lambda x.t)\bar{o}) & = & (\lambda x.F_2(t))\bar{o} & \textit{if } t \notin SN_\beta \\
F_2((\lambda x.t)s\bar{o}) & = & (\lambda x.t)F_2(s)\bar{o} & \textit{if } t \in SN_\beta, s \notin SN_\beta \\
F_2((\lambda x.t)s\bar{o}) & = & \underline{(\lambda x.t)s}\bar{o} & \textit{if } t, s \in SN_\beta
\end{array}$$

Definition 2.30 *([Sør98, VRSSX99]) The β-reduction strategy F_3 is defined as follows:*

$$\begin{array}{lcll}
F_3(x\bar{t}s\bar{o}) & = & x\bar{t}F_3(s)\bar{o} & \textit{if } \bar{t} \in SN_\beta, s \notin SN_\beta \\
F_3(\lambda x.t) & = & \lambda x.F_3(t) & \\
F_3((\lambda x.t)\bar{o}) & = & (\lambda x.F_3(t))\bar{o} & \textit{if } t \notin SN_\beta \\
F_3((\lambda x.t)\bar{o}s\bar{e}) & = & (\lambda x.t)\bar{o}F_3(s)\bar{e} & \textit{if } t, \bar{o} \in SN_\beta, s \notin SN_\beta \\
F_3((\lambda x.t)s\bar{o}) & = & \underline{(\lambda x.t)s}\bar{o} & \textit{if } t, s, \bar{o} \in SN_\beta
\end{array}$$

Definition 2.31 *The β-reduction strategy F^∞_{lm} is defined as follows:*

$$\begin{array}{lcll}
F^\infty_{lm}(x\bar{t}s\bar{o}) & = & x\bar{t}F^\infty_{lm}(s)\bar{o} & \textit{if } \bar{t} \in SN_\beta, s \notin SN_\beta \\
F^\infty_{lm}(\lambda x.t) & = & \lambda x.F^\infty_{lm}(t) & \\
F^\infty_{lm}((\lambda x.t)\bar{o}) & = & (\lambda x.F^\infty_{lm}(t))\bar{o} & \textit{if } t \notin SN_\beta \\
F^\infty_{lm}((\lambda x.t)\bar{o}s\bar{e}) & = & (\lambda x.t)\bar{o}F^\infty_{lm}(s)\bar{e} & \textit{if } t, \bar{o}, (\lambda x.t)\bar{o} \in SN_\beta,\ s \notin SN_\beta \\
F^\infty_{lm}((\lambda x.t)s\bar{o}\bar{e}) & = & \underline{(\lambda x.t)s}\bar{o}\bar{e} & \textit{if } t, s, \bar{o} \in SN_\beta,\ (\lambda x.t)s\bar{o} \notin SN_\beta
\end{array}$$

Definition 2.32 *([Kha94c, Kha94a, Kha01]) The* limit *strategy F^{∞}_{lim} in an orthogonal fully-extended CCERS is defined as follows:*

1. *Let u_l be a redex in a term t defined as follows: choose an external redex u_1 in t; choose an erased argument s_1 of u_1 that is not in normal form (if any); choose in s_1 an external redex u_2, and so on as long as possible. Let $u_1, s_1, u_2, \ldots, u_l$ be such a sequence. The redex u_l is called a* limit redex *of t.*

2. *We call a strategy* limit*, noted F^{∞}_{lim}, if in any term not in normal form it selects a limit redex. (Note that by Theorem 2.19 in any term not in normal form there is a limit redex.)*

2.4 Two characterizations of critical redexes

In this section we give an intuitive characterization of critical redex occurrences for orthogonal fully-extended CCERSs, generalizing Klop's characterization of critical redex occurrences for orthogonal TRSs [Klo92], and derive from it a characterization of perpetual redexes similar to Bergstra and Klop's perpetuality criterion for β-redexes [BK82]. Our proofs are surprisingly simple, yet the results are rather general and useful in applications. We need three simple lemmas first.

Lemma 2.33 *Let $t \xrightarrow{u} s$ be in a CCERS, let $o \subseteq t$ be either in an argument of u or not overlapping with u, and let $o' \subseteq s$ be a u-descendant of o. Then $o' = o\theta$ for some substitution θ. Moreover, if o is a redex, then so is o' and $o \sim o'$.*

Proof Since u can be decomposed as a TRS-step followed by a number of substitution steps, it is enough to consider the cases when u is a TRS step and when it is an S-reduction step. If u is a TRS-step, or is an S-reduction step and o is not in its last argument, then o and o' coincide (hence $o \sim o'$ when o is a redex). Otherwise, $o' = o\theta$ for some substitution θ, and if o is a redex, we have again $o \sim o'$ by orthogonality and Corollary 2.13, since free variables of the substituted subterms cannot be bound in $o\theta$ (by the variable convention).

Lemma 2.34 *Let s be a minimal perpetual subterm of t, in an orthogonal fully-extended CCERS, and let $P : t \twoheadrightarrow \infty$ be internal to s. Then P has the form $P = t \twoheadrightarrow o \xrightarrow{u} e \twoheadrightarrow \infty$, where u is the descendant of s in o (i.e., a descendant of s necessarily becomes a redex and is contracted in P).*

Proof If P did not contract descendants of s, then infinitely many steps of P would be contracted in at least one of the proper subterms of s, and this would contradict the minimality of s.

Lemma 2.35 *In an orthogonal fully-extended CCERS, let $P = u + P'$ be a constricting minimal perpetual reduction starting from t, and let u be in an argument o of a redex $v \subseteq t$. Then P is internal to o.*

Proof Let $s \subseteq t$ be the minimal perpetual subterm containing u. By definition of minimal perpetual reductions, u is an external redex of s; hence s does not contain v. Since P is constricting, it is internal to s, and orthogonality and Lemma 2.34 tell us that s cannot overlap the pattern of v. The lemma follows.

Definition 2.36 *(1) Let $P : t_0 \xrightarrow{u_0} t_1 \xrightarrow{u_1} \ldots \xrightarrow{u_{k-1}} t_k$, be in an orthogonal CCERS, and let $s_0, s_1, \ldots, s_k$ be a chain of descendants of s_0 along P (i.e, s_{i+1} is a u_i-descendant of $s_i \subseteq t_i$). Then, following [BK82], we call P* passive *w.r.t. $s_0, s_1, \ldots, s_k$ if the pattern of u_i does not overlap s_i (s_i may be in an argument of u_i or be disjoint from u_i) for $0 \le i < k$, and we call s_k a* passive *descendant of s_0. By Lemma 2.33, $s_k = s\theta$ for some substitution θ, which we call a* passive substitution, *or* P-substitution *(w.r.t. $s_0, s_1, \ldots, s_k$).*

(2) Let t be a term in an orthogonal fully-extended CCERS and let $s \subseteq t$. We call s a potentially infinite *subterm of t if s has a passive descendant s' s.t. $\infty(s')$. (Thus $\infty(s\theta)$ for some passive substitution θ.)*

Theorem 2.37 *Let t be an ∞-term and let $t \xrightarrow{v} s$ be a critical step in an orthogonal fully-extended CCERS. Then v erases a potentially infinite argument o (thus $\infty(o\theta)$ for some passive substitution θ).*

Proof Let $P : t = t_0 \xrightarrow{u_0} t_1 \xrightarrow{u_1} t_2 \twoheadrightarrow \infty$ be a constricting minimal perpetual reduction, which exists by Theorem 2.23 and Theorem 2.25. Since v is critical, $SN(s)$; hence P/v is finite. Let j be the minimal number such that $u_j/V_j = \emptyset$ and $u_j \notin V_j$, where $V_j = v/P_j$ and $P_j : t \twoheadrightarrow t_j$ is the initial part of P with j steps. (Below, V_j will denote both the corresponding set of residuals of v and its complete development.) By the Finite Developments theorem, no tails of P can contract only residuals of v; and since P/v is finite, such a j exists.

$$\begin{tikzcd}
t = t_0 \arrow[r, two heads] \arrow[d, "v"'] & t_l \arrow[r, two heads] \arrow[d, two heads, "V_l"'] & t_j \arrow[r, "u_j"] \arrow[d, two heads, "V_j"'] & t_{j+1} \arrow[r, two heads] \arrow[d, two heads] & P \\
s = s_0 \arrow[r, two heads] & s_l \arrow[r, two heads] & s_j \arrow[r, two heads, "\emptyset"'] & s_{j+1} \arrow[r, two heads, "\emptyset"'] & P/v
\end{tikzcd}$$

Since $u_j/V_j = \emptyset$ and $u_j \notin V_j$, there is a redex $v' \in V_j$ whose residual is contracted in V_j and erases (the residuals of) u_j. Since V_j consists of (possibly nested) residuals of a single redex $v \subseteq t_0$, the quantifiers in the pattern of v' cannot bind variables inside arguments of other redexes in V_j. Therefore, by Corollary 2.13, v' is similar to its residual contracted in V_j, and hence $u_j/v' = \emptyset$, implying that v' erases its argument o', say the m-th from the left, containing u_j. By Lemma 2.35, the tail $P_j^* : t_j \twoheadrightarrow \infty$ of P is internal to o'.

Let $v_i \subseteq t_i$ be the predecessors of v' along P_j (so $v_0 = v$ and $v_j = v'$; note that a redex can have at most one predecessor), and let o_i be the m-th argument of v_i (thus $o' = o_j$). Note that $u_i \neq v_i$ because v_i has residuals. Let l be the minimal number such that u_l is in an argument of v_l (such an l exists because u_j is in an argument of v_j). Then, by Lemma 2.35, all the remaining steps of P are in the same argument of v_l and it must be the m-th argument o_l of v_l (thus $\infty(o_l)$); but v' erases its m-th argument, implying by

Corollary 2.13 that v_l also erases its m-th argument o_l. Furthermore, by the choice of l, no steps of P are contracted inside v_i for $0 \leq i < l$; thus v_l is a passive descendant of v, and o_l is a passive descendant of o_0. Hence, by Lemma 2.33 $v \sim v_l$. Thus v erases a potentially infinite argument o_0 (since $\infty(o_l)$), and we are done.

Note in the above theorem that if the orthogonal fully-extended CCERS is an orthogonal TRS, a potentially infinite argument is actually an ∞-term (since passive descendants are all identical), implying Klop's perpetuality lemma [Klo92]. O'Donnell's [O'D77] lemma, stating that any term from which an innermost reduction is normalizing is strongly normalizing, is an immediate consequence of Klop's Lemma.

Corollary 2.38 *Any redex whose erased arguments are closed SN terms is perpetual in orthogonal fully-extended CCERSs.*

Proof Immediate, since closed SN terms cannot be potentially infinite subterms.

Note that Theorem 2.37 implies a general (although not computable) perpetual strategy: simply contract a redex u in the term t whose erased arguments (if any) are not potentially infinite w.r.t. at least one ∞-subterm $s \subseteq t$ (although the erased arguments of u may be potentially infinite w.r.t. t). It is easy to see that the perpetual strategy F^{∞} of Barendregt et al. [BBKH76, Bar84] and, in general, the *limit* perpetual strategy F^{∞}_{lim} of Khasidashvili [Kha94c, Kha94a, Kha01] are special cases, since these strategies contract redexes whose arguments are in normal form and no (sub)terms can be substituted in the descendants of these arguments. The strategy F^{∞}_{m} (and hence the strategies F_3 and F_z), as well as the strategies F_1 and F_2, are also special cases of the above general perpetual strategy.

We conclude this section with a characterization of the perpetuality of erasing redexes, a characterization similar to the perpetuality criterion of β_K-redexes that was given by Bergstra and Klop [BK82].

Below, a substitution θ will be called SN iff $SN(x\theta)$ for every variable x.

Definition 2.39 *We call a redex u* safe *(respectively,* SN-safe*) if it is non-erasing or if it is erasing and for any (resp. SN-) substitution θ, if $u\theta$ erases an ∞-argument, then the contractum of $u\theta$ is an ∞-term. (Note that, by Corollary 2.13, u is erasing iff $u\theta$ is, for any θ, erasing.)*

Theorem 2.40 *In an orthogonal fully-extended CCERS R, any safe redex v is perpetual.*

Proof Assume on the contrary that there is a context $C[\,]$ such that $t = C[v] \to s$ is a critical step. Let l be the minimal number such that, for some constricting minimal perpetual reduction $P : t = t_0 \xrightarrow{u_0} t_1 \xrightarrow{u_1} t_2 \twoheadrightarrow \infty$, the tail $P^*_l : t_l \twoheadrightarrow \infty$ of P is in an erased argument of a residual of v. Such an l exists by the proof of Theorem 2.37 (in the notation of that theorem, P^*_l is in an erased argument of $v_l \subseteq t_l$). Let v_l be the outermost of the redexes in t_l which contain u_l (and therefore, P^*_l) in an erased argument o_l, say the m-th from the left (thus $\infty(o_l)$). By the proof of Theorem 2.37, the m-th argument o of v is v-erased, $o_l = o\theta$, and $v_l = v\theta$ for some passive substitution θ.

We want to prove that the safety of v implies $\infty(s_l)$, hence $\infty(s)$, contradicting the assumption that $t \xrightarrow{v} s$ is critical (see the diagram for Theorem 2.37). By the Finite

Developments theorem, we can assume that s_l is obtained from t_l by contracting (some of) the redexes in V_l in the following order: (a) contract redexes in V_l disjoint from v_l; (b) contract redexes in V_l that are in the main arguments of v_l; (c) contract the residual v_l^* of v_l; (d) contract the remaining redexes, i.e., those containing v_l in a main (by the choice of v_l) argument. Since the contractions (a) and (b) do not affect o_l, v_l^* erases an ∞-argument. (Recall from the proof of Theorem 2.37 that redexes in V_l are similar to their residuals contracted in any development of V_l.) Since $v_l = v\theta$ and redexes in (b) are in the substitution part of v_l, $v_l^* = v\theta^*$ for some substitution θ^*; hence its contractum e is infinite by the safety of v. By the choice of v_l, e has a descendant e' in s_l after the contractions (d). By the following diagram (where t_l^0 is obtained from t_l by the steps (a), (b) and (c); $w_0 + w_1 + \ldots$ is an infinite reduction of $e \subseteq t_l^0$; $U_0^{(d)}$ is the set of residuals of redexes in (d); and $U_i^{(d)}$ are respective residuals of $U_0^{(d)}$), $\infty(e)$ implies $\infty(e')$. Indeed, if $e_i \subseteq t_l^i$ is the descendant of e in t_l^i, then all $U_i^{(d)}$-descendants of e_i in s_l^i are disjoint and identical to e_i, and $s_l^i \twoheadrightarrow s_l^{i+1}$ contracts exactly one residual of w_i in every $U_i^{(d)}$-descendant of e_i (the latter are also descendants of $e' \subseteq s_l$). Hence $\infty(s_l)$ – a contradiction.

$$\begin{tikzcd}
t_l^0 \arrow[r, "w_0"] \arrow[d, two heads, "U_0^{(d)}"'] & t_l^1 \arrow[r, "w_1"] \arrow[d, two heads, "U_1^{(d)}"'] & t_l^2 \arrow[r, "w_2"] \arrow[d, two heads, "U_2^{(d)}"'] & t_l^3 \arrow[r, two heads] \arrow[d, two heads, "U_3^{(d)}"'] & {} \\
s_l = s_l^0 \arrow[r, two heads, "+"'] & s_l^1 \arrow[r, two heads, "+"'] & s_l^2 \arrow[r, two heads, "+"'] & s_l^3 \arrow[r, two heads, "+"'] & {}
\end{tikzcd}$$

Møller Neergaard and Sørensen [MNS99] give a different proof of perpetuality of safe K-redexes in the λ-calculus (safe K-redexes are called there *good*).

The following example demonstrates that non-erasing steps need not be perpetual in orthogonal CCERSs in general, that is, the restriction to fully-extended CCERSs is necessary:

Example 2.41 *Consider the ERS with rules:*

$$\begin{array}{rcl}
\lambda x(A, B) & \to & (B/x)A \\
\kappa yz(A) & \to & (a/z)A \\
e(A, B) & \to & c \\
f(a) & \to & f(a)
\end{array}$$

where λ is a partial quantifier symbol binding only in its first argument, and $y \notin FV(A\theta)$ for any assignment θ admissible for the κ-rule. Consider the term $s = \kappa yz(\lambda x(e(x, y), f(z)))$. Note that s is not a redex (yet) due to the occurrence of y. On the one hand, contracting the e-redex yields an infinite reduction

$$s \to \kappa yz(\lambda x(c, f(z))) \to \lambda x(c, f(a)) \to \ldots$$

On the other hand, contracting the (non-erasing) λ-redex yields

$$s \to \kappa yz(e(f(z), y)) \to \kappa yz(c) \to c$$

as only, and strongly normalizing, reduction. Hence the λ-step is non-erasing but critical.

2.5 Applications

We now give a number of applications demonstrating the power and usefulness of our perpetuality criteria. In some of the examples we will use the conventional λ-calculus notation [Bar84], and by the *argument* of a β-redex $(\lambda x.s)o$ we will mean its second argument o.

2.5.1 The restricted orthogonal λ-calculi

Let us call an *orthogonal restricted λ-calculus* (ORLC) a calculus that is obtained from the λ-calculus by restricting the term set and the β-rule (by some conditions on arguments and contexts) and that is an orthogonal fully-extended CCERS. Examples include the λ_I-calculus, the call-by-value λ-calculus [Plo75], and a large class of typed λ-calculi.

If R is an ORLC, then in the proofs of Theorem 2.37 and Theorem 2.40, the P_l-substitution (and in general, any passive substitution along a constricting perpetual reduction) is SN. This can be proved in a way similar to the one used to prove the Bergstra-Klop criterion (see [BK82, Proposition 2.8]), since in the terminology of [BK82] and in the notation of Theorem 2.37 and Theorem 2.40:

- P_l is SN-substituting (meaning that the arguments of contracted β-redexes are SN). This is immediate from the minimality of P_l.

- P_l is *simple* (meaning that no subterms can be substituted in the subterms substituted during the previous steps). This follows immediately from externality, w.r.t. the chosen minimal perpetual subterm, of minimal perpetual redexes (P_l is standard).

Hence, we have the following two corollaries. The first one is a perpetuality criterion for redex-occurrences and can be seen as a refinement of the Bergstra-Klop criterion [BK82] in that it takes into account passive substitutions that can be generated by the context. The second corollary is simply an extension of the Bergstra-Klop criterion (in the case of β-redexes, the converse statement is much easier to prove, see [BK82]).

Corollary 2.42 *Let t be an ∞-term and let $t \xrightarrow{v} s$ be a critical step in an ORLC. Then v erases a potentially infinite argument o such that $\infty(o\theta)$ for some passive SN-substitution θ.*

Corollary 2.43 *In an ORLC, any SN-safe redex v is perpetual.*

For the case of the λ-calculus, a different proof of Corollary 2.43 was published by Xi [Xi96]. A simple proof of the Bergstra-Klop criterion, one that uses the strategy F_2 and thus is closely related to our proof was given by van Raamsdonk et al. [VRSSX99] (that proof was obtained independently). Honsell and Lenisa [HL99] derive a strengthened version of the Bergstra-Klop criterion using semantical methods. They show that β-redexes that are safe w.r.t. *closed NF-substitutions* are also perpetual (closed *NF*-substitutions instantiate variables by closed normal forms). This criterion cannot be derived (at least, directly) from the above corollaries.

Note that these corollaries are not valid for orthogonal fully-extended CCERSs in general since, unlike the passive substitutions in an ORLC, the passive substitutions along constricting perpetual reductions in orthogonal fully-extended CCERSs need not be SN: Let $R = S \cup \{\sigma xAB \to Sx\omega(A/x)B,\ E(A) \to a\}$ where $\omega = \lambda x.Ap(x,x)$. Then the step $\sigma xAp(x,x)E(x) \to \sigma xAp(x,x)a$ is SN-safe (since it erases only a variable) but is critical as can be seen from the following diagram, of which the bottom part is the only reduction starting from $\sigma xAp(x,x)a$:

$$\begin{tikzcd}
\sigma xAp(x,x)E(x) \arrow[r, "\sigma"'] \arrow[d, "E"'] & SxwE(Ap(x,x)) \arrow[r, "S"'] \arrow[d, "E"'] & E(Ap(w,w)) \arrow[r, "\beta"'] \arrow[d, "E"'] & E(Ap(w,w)) \arrow[r, "\beta"'] \arrow[d, "E"'] & {} \\
\sigma xAp(x,x)a \arrow[r, "\sigma"'] & Sxwa \arrow[r, "S"'] & a \arrow[r, two heads, "\emptyset"'] & a \arrow[r, two heads, "\emptyset"'] & {}
\end{tikzcd}$$

2.5.2 Plotkin's call-by-value λ-calculus

To investigate the relation between the λ-calculus and ISWIM language of Landin [Lan64], Plotkin [Plo75] introduced the *call-by-value* λ-calculus λ_V. This calculus restricts the λ-calculus by allowing the contraction of redexes whose arguments are *values*, i.e., either abstractions $\lambda x.t$ or variables (we assume that there are no δ-rules in the calculus). Let the *lazy* call-by-value λ-calculus λ_{LV} be obtained from λ_V by allowing only call-by-value redexes that are not in the scope of a λ-occurrence (λ_{LV} is enough for computing values in λ_V, see Corollary 1 in [Plo75]). Then it follows from Corollary 2.42, as well as from Corollary 2.43, that any λ_{LV}-redex is perpetual; hence λ_{LV} is UN. Indeed, let $v = (\lambda x.s)o$ be a λ_{LV}-redex. Then if o is a variable it is immediate that v cannot be critical and that if o is an abstraction any of its instances is an abstraction too and hence is a λ_{LV}-normal form. This is not surprising, however, because λ_{LV}-redexes are disjoint[14] and there is no duplication or erasure of (admissible) redexes.

2.5.3 De Groote's β_{IS}-reduction

De Groote [DG93] introduced β_S-reduction on λ-terms by the following rule:

$$\beta_S : ((\lambda x.M)N)O \to (\lambda x.MO)N,$$

where $x \notin FV(M,O)$. He proved that the β_{IS}-calculus is UN. Clearly, this is an immediate corollary of Theorem 2.37 since the β_S- and β_I-rules are non-erasing (note that these rules do not conflict because of the conditions on bound variables). Using this result, de Groote proves strong normalization of a number of typed λ-calculi.

2.5.4 Böhm and Intrigila's λ-δ_k-calculus

Böhm and Intrigila [BI94] introduced the λ-δ_k-calculus in order to study UN solutions to fixed point equations, in the $\lambda\eta$-calculus. Since the K-redexes are the reason for

[14] if u, v are redexes in a term t and $u = (\lambda x.e)o$, then $v \notin e$ because of the main λ of u, and $v \notin o$ since o is either a variable or an abstraction; orthogonality of λ_{LV} follows from a similar argument.

the failure of uniform normalization in the $\lambda(\eta)$-calculus, Böhm and Intrigila define a 'restricted' K-combinator δ_K by the following rule:

$$\delta_K AB \to A,$$

where B can be instantiated to closed λ-δ_k-normal forms (possibly containing δ_K constants; such a reduction is still well defined). λ-δ_k-terms are λ_I-terms with the constant δ_K. Böhm and Intrigila show that the λ-δ_k-calculus is UN.

Whereas the η-rule is not fully-extended on the set of all (possibly erasing) terms, it is fully-extended on the restricted set of (non-erasing) λ-δ_k-terms. However, UN does not follow from Corollary 2.38 since λ-δ_k-calculus violates the orthogonality assumption. It is only weakly orthogonal since there are the usual (trivial) critical pairs between the β- and η-rule. This is settled in our other paper [KOvO01b], and will be remarked in Section 2.6.

2.5.5 Honsell and Lenisa's β_{N°- and β_{KN}-calcului

Motivated by a semantical study of the λ_I- and λ_V-calculi, Honsell and Lenisa [HL93] and Lenisa [Len97a] defined similar reductions, β_{N°- and β_{KN}-reductions, respectively, on λ-terms by the following rules:

$$\beta_{N^\circ}: \;\; (\lambda x.A)B \to (B/x)A,$$

where $\theta \in AA(\beta_{N^0})$ iff $\theta(B)$ is a closed β-normal form, and

$$\beta_{KN}: \;\; (\lambda x.A)B \to (B/x)A,$$

where $\theta \in AA(\beta_{KN})$ iff either $x \in FV(A\theta)$ or $\theta(B)$ is a variable or a closed β-normal form. We have immediately from Corollary 2.38 and Corollary 2.43, respectively, that β_{N° and β_{KN} are UN. Note however that these conclusions do not follow (at least, without an extra argument) from Bergstra and Klop's or Honsell and Lenisa's characterizations of perpetual β_K-redexes [BK82, HL99], since $\beta_{N^\circ}, \beta_{KN} \subset \beta$ but not vice versa. (If t has an infinite β_{N°-reduction and $t \overset{u}{\to} s$ is a β_{N°-step, then the Bergstra-Klop and Honsell-Lenisa criteria imply the existence of an infinite *β-reduction* starting from s, not the existence of an infinite β_{N°-reduction, and similarly for β_{KN}.) In [HL93], semantical proofs of UN for β_{N° and β_{KN} are given.

2.6 Concluding remarks

We have introduced (orthogonal fully-extended) Context-sensitive Conditional Expression Reduction Systems in which several (typed or untyped) λ-calculi can be expressed straightforwardly. Furthermore, we have obtained two powerful criteria for the perpetuality of redexes in orthogonal fully-extended CCERSs and have demonstrated their usefulness in applications.

As stated above, we claim that our results are also valid for Klop's orthogonal fully-extended *substructure* CRSs [KvOvR93].

Intuitively this is the case since both ERSs and CRSs are essentially second-order frameworks, i.e., abstractions over metavariables are not allowed. We will now present an example showing that allowing abstractions on function variables, as is possible in Nipkow's higher-order rewriting systems [Nip93], renders the Conservation Theorem invalid. The example exhibits a non-erasing step which is not perpetual.

Example 2.44 *Consider the higher-order rewrite system with rules:*

$$\begin{array}{lcl} f(\lambda yz.F(\lambda x.y(x),z)) & \to_f & F(\lambda x.c,\Omega) \\ app(abs(\lambda x.F(x)),S) & \to_{beta} & F(S) \end{array}$$

where the first rule contains a function variable (y) as argument to a free variable (F), the second rule is the usual [MN98] higher-order rendering of the β-rule from λ-calculus, and $\Omega = app(abs(\lambda x.app(x,x)),abs(\lambda x.app(x,x)))$. Then

$$f(\lambda yz.app(abs(\lambda x.y(x)),z)) \to_{beta} f(\lambda yz.y(z))$$

is non-erasing but critical. This can be seen from the following diagram, of which the bottom part is the only reduction starting from $f(\lambda yz.y(z))$.

$$\begin{tikzcd}
f(\lambda yz.app(abs(\lambda x.y(x)),z)) \arrow[r, "f"'] \arrow[d, "beta"'] & app(abs(\lambda x.c),\Omega) \arrow[r, "\Omega", "beta"'] \arrow[d, "beta"'] & app(abs(\lambda x.c),\Omega) \arrow[r, "\Omega", "beta"'] \arrow[d, "beta"'] & \cdots \\
f(\lambda yz.y(z)) \arrow[r, "f"'] & c \arrow[r, two heads, "\emptyset"'] & c \arrow[r, two heads, "\emptyset"'] & \cdots
\end{tikzcd}$$

The point of the example is that, unlike in the ERS- or CRS-case, in HRSs a substitution inside (caused by contracting a redex outside) a non-erasing redex can turn it into an erasing one.

In [KOvO01b], by restricting to second-order rewriting systems, we showed that similar perpetuality criteria (Theorem 2.37) holds as well as ERSs or CRSs. Furthermore, instead of observing wheter a redex is external in all reduction sequences, we restrict ourselves to observe those in given sequences (which is the result of standardization). This enables us to loosen the orthogonality assumption to the weakly orthogonality (or further the biclosed assumption) such that **UN** of λ-δ_k-calculus [BI94] in Section 2.5.4 becomes an easy corollary.

Part II

Automatic support for hand-coded programs

Chapter 3

Analysis of monadic properties based on abstract interpretation

Many static analyses have been proposed towards efficient implementation of lazy functional languages [AH87, Nie86, PJ87, Bur91, NNH99]. The most notable class is a set of *strictness analyses* [Blo86, CPJ85, HY86, Myc80, WH87], that collect information on the strictness of functions, and this information enables us to optimize a program by the so-called *call-by-need to call-by-value transformation.* A strictness analysis detects a set of parameters that *must* be evaluated to obtain the resulting value of a function.

Abstract interpretation [AH87, CC77, MK84, Myc80] and projection analysis [WH87, DW90] have been proposed as the basis for formalizing various static analyses. Abstract interpretation executes a program for all instances on a (possibly finite) abstract domain which reflects the objective property. In contrast, projection analysis interprets a program as a transformer on a (possibly finite) selections of projections, which reflect the objective property. Equivalence between their certain sub-classes was investigated in [Bur90]. However, the following questions still remain.

- In [WH87], Wadler indicated that head-strictness detection on a nonflat domain cannot be treated by previously proposed abstract interpretation, although projection analysis can. In [Bur90], Burn divided head-strictness into two parts: H-strictness and H_B-strictness. He showed that H_B-strictness can be treated by abstract interpretation. In fact, it is much easier to detect as a by-product property in total/tail strictness analysis. Then, the question is: Can H-strictness also be treated by abstract interpretation?

- There are various algorithms for program analyses. For instance, strictness analyses on flat domains were proposed in both forward/backward manners [Myc80, PJ87, HY86]. Furthermore, a result of some analysis intuitively introduces results of the others. For instance, this is a case of CPA and strictness analyses. Then, the question is: How can the equivalence and hierarchy be formally shown?

In this chapter, first, *HOMomorphic Transformer* (HOMT) is proposed in order to formalize relations among strictness analyses on first-order functional programs. A HOMT is a composition of special instances of abstract interpretation, and has enough ability to treat known static analyses including head/tail/total strictness detection on nonflat domains. A set of HOMTs, furthermore, is an algebraic space such that some composition of HOMTs can be reduced to a simpler HOMT. This structure gives a transformational mechanism between various analyses, and further clarifies the equivalence and the hierarchy among them.

Next, *Computation Path Analysis* (CPA) is proposed. CPA detects all possible demand patterns of a function, and modes represent the level of required evaluation. CPA are compared with various analyses within the formalization by HOMTs, and concludes that CPA is the most powerful.

Finally, an application of CPA on error detection, called *anomaly detection*, and the implementation of CPA with modes are shown. For this implementation, the specification of modes, which reflects the design scheme of an abstract domain, is open to users, and the system automatically generates a CPA. The design scheme of CPAs for detecting monadic properties is also discussed.

For simplicity, the arguments are basically for strongly-typed first-order functional programs, and example programs are described by LISP-like syntax during this Chapter. (Types are often omitted, if they are clear from the context.) Note that the techniques are not exclusive of polymorphically typed or typeless programs; the important classification is among flat/nonflat domains and Boolean values, which decide the choices of conditional branches.

3.1 Anatomy of abstract interpretation - HOMomorphic Transformer (HOMT)

In this section, first, we show a construction of a HOMomorphic Transformer (HOMT) as a composition of Unit-HOMTs (U-HOMTs), which are specified by quadruplet representations. Various analyses are formalized as HOMTs in either a forward or backward manner. Second, their soundness are discussed in a uniform way. Third, reduction relation among HOMTs are shown as reduction rules among specific pairs of quadruplet representations. Thus, hierarchy among HOMTs can be clarified by finding some adequate quadruplet representation that bridges one HOMT to another.

3.1.1 Construction of U-HOMTs and their quadruplet representations

A HOMT is a functional that maps continuous functions on power sets of domains to those on power sets of possibly finite abstract power domains. A HOMT is constructed as a composition of U-HOMTs. Let us describe them formally.

A quasi order $\sqsubseteq$ is a binary relation that satisfies reflexivity (i.e. $X \sqsubseteq X$) and transitivity (i.e. $X \sqsubseteq Y$ and $Y \sqsubseteq Z$ imply $X \sqsubseteq Z$), but may not satisfies anti-symmetry (i.e. $X \sqsubseteq Y$ and $X \sqsupseteq Y$ imply $X = Y$).

Definition 3.1 *Let* $(D, \sqsubseteq)$ *be a partially ordered set. A subset* $X \subseteq D$ *is* directed *if* $X \neq \phi$ *and* $\forall x, y \in D \exists z \in D . x \sqsubseteq z \ \wedge \ y \sqsubseteq z$.

$(D, \sqsubseteq)$ is a complete partial order *(CPO) if*

1. *There exists the least element $\bot$ $(\in D)$.*
2. *For each directed set $X(\subseteq D)$, the supremum $\sqcup X(\in D)$ exists.*

For CPOs $(D, \sqsubseteq)$, $(D', \sqsubseteq')$, a function $f : D \to D'$ is monotonic *if $f(x) \sqsubseteq' f(y)$ for each $x \sqsubseteq y$. a function $f : D \to D'$ is* continuous *if $f(\sqcup X) = \sqcup f(X)$ for each directed set $X(\subseteq D)$.*

Definition 3.2 *Let D be a CPO [Bar84]. A power domain $PD[D]$ is a quotient of $P[D] = \{X | X(\neq \phi) \subseteq D\}$ by $\sqsubseteq \cap \sqsupseteq$ for a quasi order $\sqsubseteq$ [Plo76, Smy78]. We define*

$$\begin{array}{lcl lcl}
RC(X) & = & \{x \in D \mid \exists y \in X.y \sqsubseteq x\} & Min(X) & = & \{x \in X \mid \neg\exists y \in X.y \sqsubset x\} \\
LC(X) & = & \{x \in D \mid \exists y \in X.x \sqsubseteq y\} & Max(X) & = & \{x \in X \mid \neg\exists y \in X.x \sqsubset y\} \\
Conv(X) & = & LC(X) \cap RC(X) & Conv(X) & = & LC(X) \cap RC(X)
\end{array}$$

and

$$\begin{array}{lcl}
X \sqsubseteq_0 Y & \Leftrightarrow & RC(X) \supseteq RC(Y) \\
X \sqsubseteq_1 Y & \Leftrightarrow & LC(X) \subseteq LC(Y) \\
X \sqsubseteq_{EM} Y & \Leftrightarrow & X \sqsubseteq_0 Y \ \wedge \ X \sqsubseteq_1 Y
\end{array}$$

For $\sqsubseteq_0$, $\sqsubseteq_1$, and $\sqsubseteq_{EM}$, a pair $(cl(X), rep(X))$ of a closure function and a representative function is $(RC(X), Min(X))$, $(LC(X), Max(X))$, or $(Conv(X), Conv(X))$, respectively. We also identify $PD[D]$ with $P^{rep}[D] = \{rep(X) | X(\neq \phi) \subseteq D\}$.

Definition 3.3 *The quasi order $\sqsubseteq_{-*}$ is $X \sqsubseteq_{-*} Y$ if and only if $X \sqsupseteq_* Y$ where $\sqsubseteq_*$ is either $\sqsubseteq_0$, $\sqsubseteq_1$, or $\sqsubseteq_{EM}$.*

Definition 3.4 *Let D and Abs be CPOs. We denote power sets of D (resp. Abs) by $P[D]$ (resp. $P[Abs]$).*

A map $abs : P[D] \to P[Abs]$ is a domain abstraction if abs is bottom-reflecting [Bur90] (i.e. $abs(\{x\}) = \{\bot_{Abs}\}$ implies $x = \bot_D$). A map $conc : P[Abs] \to P[D]$ is a domain concretization if $abs \cdot conc = id$. They are called necessary *if $conc \cdot abs \supseteq id$, and* sufficient *if $conc \cdot abs \subseteq id$.*

Let D_1, D_2 be sets A map $f\colon P[D_1] \to P[D_2]$ is centerized *if $f(X) = \cup_{x \in X} f(\{x\})$. A map $f\colon P[D_1] \to P[D_2]$ is a* lifting *of a map $f'\colon D_1 \to D_2$ if $f(X) = \{f'(x) \mid x \in X\}$.*

Remark A necessary pair composed of an abstraction *abs* and concretization *conc* is frequently called a *Galois connection.* For a necessary pair, the typical choice of *conc* is the inverse abs^{-1}, as in Chapter 3. In Chapter 4, the example of a sufficient pair will be presented (i.e., *conc* other than abs^{-1}).

Frequently in literature, an abstraction *abs* is a map from a domain D to an abstract domain Abs. Our definition covers them by regarding an abstraction as its lifting. Actually, a centerized abstraction (in our definition) coincides with the conventional definition. The reason why we use an abstraction $P[D] \to P[Abs]$ is the construction of sufficient pairs, which have been neglected in literature.

$$\begin{tikzcd}
D^n \arrow[r, "f"] \arrow[dr, dashed] & D \arrow[d, dashed, "\cap"] \\
P[D^n] \arrow[r, "f"] \arrow[d, shift right, "abs"'] \arrow[dr, phantom, "\circlearrowright"] & P[D] \arrow[d, "abs"] \\
P[Abs^n] \arrow[r] \arrow[u, shift right, "conc"'] \arrow[dr, phantom, "\circlearrowright"] & P[Abs] \arrow[d, "rep"] \\
PD[Abs^n] \arrow[u, "cl"] \arrow[r, "h(f)"'] & PD[Abs]
\end{tikzcd}$$

forward U-HOMT h

$$\begin{tikzcd}
D^n \arrow[r, "f"] \arrow[d, dashed, "\cap"] & D \arrow[dl, dashed] \\
P[D^n] \arrow[d, "abs"] \arrow[dr, phantom, "\circlearrowleft"] & P[D] \arrow[l, "f^{-1}"'] \arrow[d, shift right, "abs"'] \\
P[Abs^n] \arrow[d, "rep"'] \arrow[dr, phantom, "\circlearrowleft"] & P[Abs] \arrow[l] \arrow[u, shift right, "conc"'] \\
PD[Abs^n] & PD[Abs] \arrow[l, "h(f)"] \arrow[u, "cl"']
\end{tikzcd}$$

backward U-HOMT h

Figure 3.1: Construction of U-HOMT h

A U-HOMT, which is a HOMT induced from a domain abstraction and a concretization, is classified into two types: *forward U-HOMTs* and *backward U-HOMTs*. They correspond to *forward analyses* and *backward analyses*. A forward U-HOMT transforms a function to that of the same direction, and a backward U-HOMT transforms it to that of the opposite direction. Their constructions are shown in commutative diagrams in Fig. 3.1. In Fig. 3.1, *abs* is an abstraction map, *conc* is a concretization map, *cl* is a closure map, *proj* is a projection (from a power set to a power domain), and *rep* is a representative function.

U-HOMTs are constructed in following steps. For a forward U-HOMT, first, a function $f : D^n \to D$ is naturally lifted to a function $f : P[D^n] \to P[D]$ as $f(X) = \{f(x) \mid x \in X\}$, where $P[D^n]$ and $P[D]$ are power sets of D^n and D, respectively. Next, f is transformed to a function $abs \cdot f \cdot conc : P[Abs^n] \to P[Abs]$. Then, $h(f) : PD[Abs^n] \to PD[Abs]$ is obtained by $rep \cdot abs \cdot f \cdot conc \cdot cl$. By regarding $PD[Abs^n]$ as $P^{rep}[Abs^n]$ and $PD[Abs]$ as $P^{rep}[Abs]$, the interface of composition of U-HOMTs is kept as power sets, which will be introduced in Section 3.1.3. On the other hand, a backward U-HOMT h is $h(f) = rep \cdot abs \cdot f^{-1} \cdot conc \cdot cl$, where $f^{-1}(Y) = \{x \mid f(x) \in Y\}$. Then the algorithm of analysis or verification is formalized as the least-fixed-point computation on power domains, i.e., start with the minimal element in a power domain, and iteratively unfold function definitions until convergence.

Formally, there are several differences between constructions of forward and backward U-HOMTs. A result of a backward U-HOMT $h(f)$ is induced from an inverse relation f^{-1}, whereas a result of a forward U-HOMT is induced from a relation f itself. Therefore, the function inverse may require the empty set ϕ as a result value when f is not an onto relation. Thus, a power domain $PD[D]$ must be extended with ϕ to make $h(f)$ well-defined. The quasi orders $\sqsubseteq_0$, $\sqsubseteq_1$ are also naturally extended so that ϕ is largest wrt $\sqsubseteq_0$

and ϕ is least wrt $\sqsubseteq_1$.[1]

Further, for a forward case the more informative function returns the more informative output for the same input, whereas for a backward case the more informative function requires the less informative input for the same output. This means that backward abstract interpretations use the inverted quasi orders, such as $\sqsubseteq_{-0}$ ($= \sqsupseteq_0$) (e.g., strictness analysis in [HY86, Bur90, HL94]) and $\sqsubseteq_{-1}$ ($= \sqsupseteq_1$) (e.g., verification in [Oga99]).

Therefore, the parameters that specify these U-HOMT constructions are a pair composed of domain abstraction and concretization, a direction (that is, either forward or backward), a power domain construction (a pair composed of a quasi order and a closure function), and a representative function[2] By definition, a power domain construction is composed of the selection of quasi order $\sqsubseteq$ on power set combined with a closure function *closure*. Thus, once a domain abstraction is fixed there are relatively small choices.

Definition 3.5 *A U-HOMT, h, is specified by a domain abstraction abs: $P[D] \to P[Abs]$ and a domain concretization $conc$: $P[Abs] \to P[D]$, a direction dir (whether forward (+) or backward (−)), a power domain construction (i.e., a pair of a quasi order and a closure function) $(\sqsubseteq, cl)$, and a representative function rep.*

A quadruplet $((abs, conc), dir, (\sqsubseteq, cl), rep)$ is called the quadruplet representation.

There could be many choices, but in practice U-HOMTs are classifies into following cases and the power domain construction is specifically decided for each case:

	necessary	sufficient
forward	$(\sqsubseteq_1, LC)$	$(\sqsubseteq_0, RC)$
backward	$(\sqsubseteq_{-0}, RC)$	$(\sqsubseteq_{-1}, LC)$

Note that even if the framework of an abstract interpretation is determined, still the design of the abstraction and concretization is difficult. Below are some examples from literature.

- Strictness analyses over flat domains use the abstract domain $Abs = \{\mathbf{0}, \mathbf{1}\}$ with $\mathbf{0} \sqsubseteq \mathbf{1}$, the (centerized) abstraction map, $abs(\{\bot\}) = \{\mathbf{0}\}$, $abs(\{x\}) = \{\mathbf{1}\}$ for $x \in D \setminus \{\bot\}$, and the concretization $conc = abs^{-1}$.

  Then a forward strictness analysis is obtained with the power domain construction $(RC(X), \sqsubseteq_1)$ and the representative function Max, and a backward strictness analysis is obtained with $(LC(X), \sqsubseteq_{-0})$ and Min (See below).[3]

- A backward compile-time verification (in Chapter 4) uses the abstract domain $Pred$, which consists of a certain class of predicates with $P \sqsubseteq Q$ if and only if $Q \Rightarrow P$, the abstraction map abs, concretization map $conc$,

$$\begin{array}{rcl} abs(X) & = & \vee_{conc(P) \subseteq LC(X)} P \ \vee \ \mathbf{false} \\ conc(P) & = & LC(\{x \in D \mid \ P(x)\}) \end{array}$$

[1]This means that the power domain construction with $(RC(X) \cap LC(X), \sqsubseteq_0 \cap \sqsubseteq_1)$ can be applied only for a forward U-HOMT such as in [MK84].

[2]Here, we consider only the first-order abstractions; for a higher-order abstraction, refer [Hun91].

[3]Here, Min and Max are implicitly assumed to be well-defined over abstract domains. This trivially holds if an abstract domain is finite.

and the power domain construction $(RC(X), \sqsubseteq_{-1})$. Note that this verification assumes termination of a program (i.e., defined values are mapped to defined values); thus, this is partial correctness.

To conclude this section, strictness analyses are formalized in terms of HOMTs. introduce strictness analyses more formally. A strictness analysis may be either forward or backward. A strictness analysis is forward, if it clarifies the properties of results from the properties of parameters. Conversely, a strictness analysis is backward, if it clarifies the conditions satisfied by parameters from the properties of results. For simplicity, here we concentrate on strictness analyses on flat domains (i.e. *Integer*, *Boolean*, etc).

Example 3.6 *Forward Strictness Analysis (FSA) is an example of a forward analysis [Myc80]. Similar algorithms are found in [AH87, CPJ85]. FSA interprets a function, f, on a flat domain (such as* Integer *or,* Boolean *) to a $\{\mathbf{0},\mathbf{1}\}$-valued function f_{FSA}, where $\mathbf{0}$ means totally undefined and $\mathbf{1}$ means possibly defined. Thus, f_{FSA} returns $\mathbf{1}$ if there possibly exists a computable real instance of f, and returns $\mathbf{0}$ if there never exists a computable real instance of f. For instance, $if(x,y,z)$ is interpreted to*

$$\begin{array}{rllllllllll} if_{FSA}: & (\mathbf{1},\mathbf{1},\mathbf{1}) & \to & \mathbf{1}, & (\mathbf{1},\mathbf{1},\mathbf{0}) & \to & \mathbf{1}, & (\mathbf{1},\mathbf{0},\mathbf{1}) & \to & \mathbf{1}, \\ & (\mathbf{0},\mathbf{1},\mathbf{1}) & \to & \mathbf{0}, & (\mathbf{1},\mathbf{0},\mathbf{0}) & \to & \mathbf{0}, & (\mathbf{0},\mathbf{1},\mathbf{0}) & \to & \mathbf{0}, \\ & (\mathbf{0},\mathbf{0},\mathbf{1}) & \to & \mathbf{0}, & (\mathbf{0},\mathbf{0},\mathbf{0}) & \to & \mathbf{0}. & & & \end{array}$$

*Then, requisite parameters can be detected by firstly testing f_{FSA} for all $\{\mathbf{0},\mathbf{1}\}$-input patterns, next collecting the set of minimum input patterns that returns $\mathbf{1}$ (called $\mathbf{1}$-*frontier *in [AH87]), and finally detecting requisite parameters that are always required to be $\mathbf{1}$ in all patterns in the $\mathbf{1}$-frontier. For instance, $if(x,y,z)$, the $\mathbf{1}$-frontier is $\{(\mathbf{1},\mathbf{1},\mathbf{0}),(\mathbf{1},\mathbf{0},\mathbf{1})\}$, and then, the requisite parameter is x.*

A quadruplet representation of FSA is $q_{FSA} = ((abs_b, abs_b^{-1}), +, (\sqsubseteq_1, LC), Max)$. For example, $if(x,y,z)$ are interpreted to by h_{FSA} as

$$\begin{array}{rllllll} h_{FSA}(if): & Max(\{(\mathbf{1},\mathbf{1},\mathbf{1})\}) & \to & Max(\{\mathbf{1}\}), & Max(\{(\mathbf{1},\mathbf{1},\mathbf{0})\}) & \to & Max(\{\mathbf{1}\}), \\ & Max(\{(\mathbf{1},\mathbf{0},\mathbf{1})\}) & \to & Max(\{\mathbf{1}\}), & Max(\{(\mathbf{0},\mathbf{1},\mathbf{1})\}) & \to & Max(\{\mathbf{0}\}), \\ & Max(\{(\mathbf{1},\mathbf{0},\mathbf{0})\}) & \to & Max(\{\mathbf{0}\}), & Max(\{(\mathbf{0},\mathbf{1},\mathbf{0})\}) & \to & Max(\{\mathbf{0}\}), \\ & Max(\{(\mathbf{0},\mathbf{0},\mathbf{1})\}) & \to & Max(\{\mathbf{0}\}), & Max(\{(\mathbf{0},\mathbf{0},\mathbf{0})\}) & \to & Max(\{\mathbf{0}\}). \end{array}$$

Thus, equivalence among if_{FSA} and $h_{FSA}(if)$ is easily shown from an embedding $x \in \{\mathbf{0},\mathbf{1}\}$ to $\{x\} \in PD(\{\mathbf{0},\mathbf{1}\})$. Equivalence of quasi orders follows from $x \sqsubseteq y \Rightarrow \{x\} \sqsubseteq_1 \{y\}$.

Example 3.7 *On the other hand, Boolean-algebraic Strictness Analysis (BSA) [HY86, Ono88] is an example of a backward analysis.*[4] *BSA interprets a function, f, to a function f_{BSA} which is a symbolic manipulation on the set-characteristic expressions of input parameters. For example, $if(x,y,z)$ is interpreted to $if_{BSA}(x,y,z) = \lambda xyz.(x \cup y) \cap (x \cup z)$.*

[4]If we put *id* as an abstraction map in SIA, we will obtain *Inverse Image Analysis* [Dyb91].

Then, requisite parameters are collected by substituting actual variable names to corresponding set-characteristic expressions. For instance, requisite parameters of $if(a,b,b)$ *are detected as* $(\ \lambda xyz.(x \cup y) \cap (x \cup z)\ )(\{a\},\{b\},\{b\}) = (\{a\} \cup \{b\}) \cap (\{a\} \cup \{b\}) = \{a,b\}$.

Both FSA and BSA have equivalent analytical power except that FSA can detect diverged functions when its **1***-frontier is an empty set, whereas BSA cannot. By adding the special element* $\#$ *that represents the divergence with rules* $x \cap \# = \#$ *and* $x \cup \# = x$*, and setting the initial value of the algorithm as* $\#$ *(instead of a strict function), we obtain the backward analysis with the same analytical power, called Strictness Information Analysis (SIA) [Ono88].*

SIA, the extension of BSA, is represented as a HOMT h_{SIA} *of a quadruplet representation is* $((abs_b, abs_b^{-1}), -, (\sqsubseteq_{-0}, RC), Min)$*. The main difference between BSA and SIA is that SIA can detect diverged functions whereas BSA cannot. That is, a diverged function is interpreted as* $\lambda x_1 \ldots x_n.\#$ *by SIA whereas* $\lambda x_1 \ldots x_n.x_1 \cup \ldots \cup x_n$ *(strict function) by BSA because of the initial value of iterative algorithms. Other primitive functions are interpreted to the same functions on abstract domains. For example,* $if(x,y,z)$ *and* $+(x,y)$ *are interpreted to*

f	$h_{BSA}(f)$	$h_{SIA}(f)$
$if(x,y,z)$	$\lambda xyz.(x \cup y) \cap (x \cup z)$	$Min(\{\mathbf{1}\}) \rightarrow Min(\{(\mathbf{1},\mathbf{1},\mathbf{1}),(\mathbf{1},\mathbf{1},\mathbf{0}),(\mathbf{1},\mathbf{0},\mathbf{1})\})$
$+(x,y)$	$\lambda xy.x \cup y$	$Min(\{\mathbf{1}\}) \rightarrow Min(\{(\mathbf{1},\mathbf{1})\})$

As shown in examples above, a backward analysis can provide the demand-driven detection, whereas a forward analysis needs to compute all possibility. This is because the property is usually described as *whether the result is such and such*, etc., and an analysis needs to detect *what kinds of inputs will lead some specific result value.* Thus, it is frequently said that a backward analysis is more efficient than a forward one if they have same analyzing power. This folklore will be discussed again in Section 4.4, next Chapter.

3.1.2 Soundness of a HOMT

A U-HOMT $h(f)$ that reflects a run-time property of a function is not computable, even if the abstract domain *Abs* is finite. Therefore, instead of $h(f)$, we compute $h^c(f)$ by the least fixed point computation over an (possibly finite) abstract domain. Thus, $h^c(f)$ is an approximation of $h(f)$, and validity of the analysis depends on whether $h^c(f)$ is sound for $h(f)$.

Definition 3.8 *Let* f *be a* $fix(\tau) \equiv \sqcup_n \tau^n(\Omega)$*, for a recursion equation* τ *and an undefined function* Ω *which maps every elements to the bottom element* $\perp$*. Let* h *be a forward (resp. backward) HOMT. A computed HOMT* h^c *is* $h^c(f) \overset{\text{def}}{=} \sqcup_n (h_\tau(\tau))^n(h(\Omega))$*, where* $h_\tau(\tau)$ *is a syntactically identical (resp. inverse) equation, and interprets each primitive function* $priv$ *to* $h(priv)$*.*

Thus, $h^c(f)$ is a result of a fixed-point computation on a power domain of an abstract domain, starting with $h(\Omega)$. From the viewpoint of annalyses as HOMTs, a computed

HOMT h^c must *properly* approximates a HOMT h. For instance, consider strictness analysis, relevance analysis (which detects a set of parameters that *may* be evaluated), and CPA (which detects a set of all possible demand patterns called PDPS, see Section 3.2) on flat domains. They must satisfy the conditions shown in Table 3.1. These conditions are generalized to the *soundness* condition as defined below.

Table 3.1: Safeness of SRAs on flat domains

	real property		**detected property**
CPA	*real PDPS*	$\subseteq$	*detected PDPS*
strictness analysis	*real strict parameters*	$\supseteq$	*detected strict parameters*
relevance analaysis	*real relevant parameters*	$\subseteq$	*detected relevant parameters*

Definition 3.9 *An abstract interpretation h is sound if and only if $h(f) \sqsubseteq h^c(f)$ for all continuous functions f where $\sqsubseteq$ is a quasi order associated to a power domain construction of h.*

In general, soundness $h \sqsubseteq h^c$, is shown by two steps. That is, first show $h(f \cdot g) \sqsubseteq h(f) \cdot h(g)$ if h is forward, and $h(f \cdot g) \sqsubseteq h(g) \cdot h(f)$ if h is backward. Second, show $h(f \sqcup g) \sqsubseteq h(f) \sqcup h(g)$. (As the special case, $h(f \sqcup g) = h(f) \sqcup h(g)$, i.e., h is continuous.)

Then, the first condition guarantees $h(f^{(i)}) \sqsubseteq (h_\tau(\tau))^i(h(\Omega))$, and thus $\sqcup_i h(f^{(i)}) \sqsubseteq \sqcup_i (h_\tau(\tau))^i(h(\omega)) = h^c(f)$. The second condition guarantees $h(f) = h(\sqcup_i f^{(i)}) \sqsubseteq \sqcup_i h(f^{(i)})$. Therefore, soundness $h(f) \sqsubseteq h^c(f)$ is shown. The next theorem collects folklore results on soundness (for instance, case 3 appears in [MK84]).

Theorem 3.10 *Let h be a U-HOMT with a quadruplet representation $((abs, conc), dir, (\sqsubseteq, cl), rep)$. Then, h is sound if the following conditions are satisfied.*

- *The domain abstraction abs $: P[D] \to P[Abs]$ and the concretization conc $: PD[Abs] \to PD[D]$ are continuous (wrt $\sqsubseteq$).*
- *The pair $(dir, (\sqsubseteq, cl)rep)$ is either*
    1. *$(+, (\sqsubseteq_1, LC), Max)$, $(-, (\sqsubseteq_{-0}, RC), Min)$ with $conc \cdot abs \supseteq id$ (necessary),*
    2. *$(+, (\sqsubseteq_0, RC), Min)$, $(-, (\sqsubseteq_{-1}, LC), Max)$ with $conc \cdot abs \subseteq id$ (sufficient), or*
    3. *$(+, (\sqsubseteq_{EM}, Conv), Conv)$ with $conc \cdot abs \sqsupseteq_{EM} id$.*

Proof Let us consider the case of $(+, \sqsubseteq_1, Max)$. For other cases, discussion similarly proceeds except that instead of $\subseteq$, $\supseteq$ for case 2 and $\sqsupseteq_{EM}$ for case 3 are applied.

Let $f^{(i)}$ be the i-th approximation $\tau^i(\Omega)$ of the recursive equation τ of f. First, we show $h(f^{(i)}) \sqsubseteq (h(\tau))^i(h(\Omega))$ by proving $h(f \cdot g) \sqsubseteq h(f) \cdot h(g)$. Since *abs* is bottom-preserving, so *conc* is from $abs \cdot conc = id$. Therefore $h^c(\Omega) = h(\Omega)$. Since $conc \cdot abs \supseteq id$, $h(f \cdot g) \subseteq h(f) \cdot h(g)$ (resp. $h(f \cdot g) \subseteq h(g) \cdot h(f)$). Thus $h(f \cdot g) \sqsubseteq h(f) \cdot h(g)$ (resp. $h(f \cdot g) \sqsubseteq h(g) \cdot h(f)$).

Second, we show $h(f) = h(\sqcup_i f^{(i)}) = \sqcup_i h(f^{(i)})$. This holds since an abstraction $abs : D \to Abs$ and a concretization $conc : PD[Abs] \to PD[D]$ are continuous. Thus, we conclude $h(f) = h(\sqcup_i f^{(i)}) = \sqcup_i h(f^{(i)}) \sqsubseteq h^c(f) = \sqcup_i (h(\tau))^i(h(\Omega))$. ■

Example 3.11 *FSA and SIA (see Example 3.6 and 3.7) are sound from Theorem 3.10.*

When a domain abstraction is not continuous (such as *computation path analysis* introduced in Section 3.2), the next theorem is useful. This theorem is obtained by pursuing set-inclusion relations and does not depend on how a HOMT (which is a composition of U-HOMTs, defined below) is constructed.

Theorem 3.12 *A necessary HOMT h is sound if the following conditions are satisfied.*

- *If h is a forward HOMT, it satisfies $h(f \cdot g) \subseteq h(f) \cdot h(g)$. If h is a backward HOMT, it satisfies $h(f \cdot g) \subseteq h(g) \cdot h(f)$.*
- *$h(f \sqcup g) \subseteq h(f) \sqcup h(g)$ and $h(\sqcup_i f_i) \subseteq \sqcup_i h(f_i)$.*

3.1.3 Reduction relation among HOMTs

A HOMT is a composition of U-HOMTs. To define the composition, Let us first define a *necessary inverse* and *sufficient inverse.*

Definition 3.13 *Let $f : P[D_1] \to P[D_2]$. For $X \subseteq D_2$, a necessary inverse of f is $f^{-N}(X) = \{y \in D_1 \mid f(\{y\}) \cap X \neq \phi\}$, and a sufficient inverse is $f^{-S}(X) = \{y \in D_1 \mid f(\{y\}) \subseteq X\}$.*

Note that if f is a lifting of a map from D_1 to D_2, then these inverses coincide with the ordinary inverse f_{-1}.

Definition 3.14 *Let h, h' be U-HOMTs with quadruplet representations $((abs, conc), dir, (\sqsubseteq, cl), rep)$ and $((abs', conc'), dir', (\sqsubseteq', cl'), rep')$ with $abs : D_1 \to D_2$ and $abs' : D_2 \to D_3$.*
When both h and h' are either necessary or sufficient, the composition $h' \circ h$ is defined as in Figure 3.2 by regarding $PD[D_2^n]$ and $PD[D_2]$ as $P^{rep}[D_2^n]$ and $P^{rep}[D_2]$, respectively. Note that for forward/backward and backward/backward compositions, $h(f)^{-N}$ is applied when U-HOMTs are necessary, and $h(f)^{-S}$ is applied when U-HOMTs are sufficient.

Let a HOMT h be a composition of U-HOMTs $h_n \circ \cdots \circ h_1$. A HOMT is *forward* if the product of all directions in each quadruplet expression of U-HOMTs $h_1, \cdots, h_n$ is positive, and is *backward* if a product of all directions is negative. From this definition, the name of *a forward HOMT* is validated as it transforms a function to that of same direction. Similarly, the name of *a backward HOMT* is validated as it transforms a function to that of different direction.

Some compositions of U-HOMTs may eventually coincide with a simpler U-HOMT. This shows that some HOMTs may be equivalent although they have different representations as compositions of U-HOMTs. This equivalence is introduced from the next *Reduction Theorem*, and defines an algebraic structure on HOMTs. Before introducing it, we prepare two notations on quasi orders.

$$\begin{tikzcd}
P[D_1^n] \arrow[r, "f"] \arrow[d, shift right, "abs"'] \arrow[d, leftarrow, shift left, "conc"] \arrow[dr, phantom, "\circlearrowright"] & P[D_1] \arrow[d, "abs"] \\
P[D_2^n] \arrow[r] \arrow[dr, phantom, "\circlearrowright"] & P[D_2] \arrow[d, "rep"] \\
PD[D_2^n] \arrow[r, "h(f)"'] \arrow[u, "cl"] & PD[D_2] \arrow[d, dotted] \\
P[D_2^n] \arrow[u, dotted] \arrow[r, "h(f)"] \arrow[d, shift right, "abs'"'] \arrow[d, leftarrow, shift left, "conc'"] \arrow[dr, phantom, "\circlearrowright"] & P[D_2] \arrow[d, "abs'"'] \\
P[D_3^n] \arrow[r] \arrow[dr, phantom, "\circlearrowright"] & P[D_3] \arrow[d, "cl'"] \\
PD[D_3^n] \arrow[r, "h' \circ h(f)"'] \arrow[u, "rep'"] & PD[D_3]
\end{tikzcd}$$

Forward/forward composition

$$\begin{tikzcd}
P[D_1^n] \arrow[r, "f"] \arrow[d, shift right, "abs"'] \arrow[d, leftarrow, shift left, "conc"] \arrow[dr, phantom, "\circlearrowright"] & P[D_1] \arrow[d, "abs"] \\
P[D_2^n] \arrow[r] \arrow[dr, phantom, "\circlearrowright"] & P[D_2] \arrow[d, "rep"] \\
PD[D_2^n] \arrow[r, "h(f)"'] \arrow[u, "cl"] \arrow[d, dotted] & PD[D_2] \\
P[D_2^n] \arrow[d, "abs'"'] \arrow[dr, phantom, "\circlearrowleft"] & P[D_2] \arrow[u, dotted] \arrow[l, "h(f)^{-N(S)}"'] \arrow[d, shift right, "abs'"'] \arrow[d, leftarrow, shift left, "conc'"] \\
P[D_3^n] \arrow[d, "rep'"'] \arrow[dr, phantom, "\circlearrowleft"] & P[D_3] \arrow[l] \\
PD[D_3^n] & PD[D_3] \arrow[l, "h' \circ h(f)"] \arrow[u, "cl'"']
\end{tikzcd}$$

Forward/backward composition

$$\begin{tikzcd}
P[D_1^n] \arrow[r, "f"] \arrow[d, "abs"'] \arrow[dr, phantom, "\circlearrowleft"] & P[D_1] \arrow[d, shift right, "abs"'] \arrow[d, leftarrow, shift left, "conc"] \\
P[D_2^n] \arrow[d, "rep"'] \arrow[dr, phantom, "\circlearrowleft"] & P[D_2] \arrow[l] \\
PD[D_2^n] \arrow[d, dotted] & PD[D_2] \arrow[l, "h(f)"] \arrow[u, "cl"'] \\
P[D_2^n] \arrow[d, "abs'"'] \arrow[dr, phantom, "\circlearrowleft"] & P[D_2] \arrow[u, dotted] \arrow[l, "h(f)"'] \arrow[d, shift right, "abs'"'] \arrow[d, leftarrow, shift left, "conc'"] \\
P[D_3^n] \arrow[d, "rep'"'] \arrow[dr, phantom, "\circlearrowleft"] & P[D_3] \arrow[l] \\
PD[D_3^n] & PD[D_3] \arrow[l, "h' \circ h(f)"] \arrow[u, "cl'"']
\end{tikzcd}$$

Backward/forward composition

$$\begin{tikzcd}
P[D_1^n] \arrow[r, "f"] \arrow[d, "abs"'] \arrow[dr, phantom, "\circlearrowleft"] & P[D_1] \arrow[d, shift right, "abs"'] \arrow[d, leftarrow, shift left, "conc"] \\
P[D_2^n] \arrow[d, "rep"'] \arrow[dr, phantom, "\circlearrowleft"] & P[D_2] \arrow[l] \\
PD[D_2^n] & PD[D_2] \arrow[l, "h(f)"] \arrow[u, "cl"'] \arrow[d, dotted] \\
P[D_2^n] \arrow[u, dotted] \arrow[r, "h(f)^{-N(S)}"] \arrow[d, shift right, "abs'"'] \arrow[d, leftarrow, shift left, "conc'"] \arrow[dr, phantom, "\circlearrowright"] & P[D_2] \arrow[d, "abs'"] \\
P[D_3^n] \arrow[r] \arrow[dr, phantom, "\circlearrowright"] & P[D_3] \arrow[d, "cl'"] \\
PD[D_3^n] \arrow[r, "h' \circ h(f)"'] \arrow[u, "rep'"] & PD[D_3]
\end{tikzcd}$$

Backward/backward composition

Figure 3.2: Composition of U-HOMTs h, h'

$$\begin{tikzcd}
 & \sqsubseteq_{EM} \arrow[dl, dash] \arrow[dr, dash] & & \subseteq \arrow[dlll, dash] \arrow[dl, dash] \\
\sqsubseteq_0 \arrow[dr, dash] & & \sqsubseteq_1 \arrow[dl, dash] & \\
 & \sqsubseteq_{null} & &
\end{tikzcd}$$

Figure 3.3: Relation among quasi orders

Definition 3.15 *The order $\ll$ among quasi orders $\sqsubseteq$ is $\sqsubseteq' \ll \sqsubseteq$ if and only if $X \sqsubseteq Y \Rightarrow X \sqsubseteq' Y$.*

The lattice due to this ordering $\ll$ is presented in Fig. 3.3. Intuitively speaking, $\sqsubseteq' \ll \sqsubseteq$ means $\sqsubseteq$ has more detailed information than $\sqsubseteq'$.

Theorem 3.16 *Let h and h' be U-HOMTs, and $((abs, conc), dir, (\sqsubseteq, cl), rep)$ and $((abs', conc'), dir', (\sqsubseteq', cl'), rep')$ be their quadruplet representations. If one of following conditions is satisfied, a composition $h' \circ h$ is reduced to a single U-HOMT h'' as $((abs', conc'), dir', \sqsubseteq', rep') \circ ((abs, conc), dir, \sqsubseteq, rep) = ((abs' \cdot abs, conc \cdot conc'), dir' \cdot dir, (\sqsubseteq', cl'), rep')$.*

1. *$(abs', conc') = (id, id)$, $dir' = +$, and either $\sqsubseteq' \ll \sqsubseteq$ when $dir = +$, or $\sqsubseteq' \ll \sqsubseteq_-$ when $dir = -$.*
2. *$dir = dir' = +$, $\sqsubseteq = \sqsubseteq'$, and $abs, conc, abs', conc'$ are monotonic.*
3. *Assume that $abs, conc$ are monotonic and centerized. $dir' = -$, $(abs', conc') = (id, id)$, and either*
    - *h is necessary, $dir = +$, $(\sqsubseteq, \sqsubseteq') = (\sqsubseteq_{-0}, \sqsubseteq_1)$, $rep = Min$, and $rep' = Max$,*
    - *h is necessary, $dir = -$, $(\sqsubseteq, \sqsubseteq') = (\sqsubseteq_1, \sqsubseteq_{-0})$, $rep = Max$, and $rep' = Min$,*
    - *h is sufficient, $dir = +$, $(\sqsubseteq, \sqsubseteq') = (\sqsubseteq_0, \sqsubseteq_{-1})$, $rep = Min$, and $rep' = Max$,*
    - *h is sufficient, $dir = -$, $(\sqsubseteq, \sqsubseteq') = (\sqsubseteq_{-1}, \sqsubseteq_0)$, $rep = Max$, and $rep' = Min$, or*
    - *$(\sqsubseteq, \sqsubseteq') = (\subseteq, \subseteq)$.*

Proof For case 1 and 2, the proof is easy. We consider case 3, namely, h, h' are necessary, $dir = -$, $(\sqsubseteq, \sqsubseteq') = (\sqsubseteq_1, \sqsubseteq_{-0})$, $rep = Max$, and $rep' = Min$. Let $abs : P[D_1] \rightarrow P[D_2]$ and $conc : P[D_2] \rightarrow P[D_1]$. It is enough to show that each primitive function f satisfy $h' \circ h(f) = h''(f)$.

Since $abs, conc$ are centerized and f is a map, $h' \circ h(f)$ and $h''(f)$ are also centerized. Thus we consider $h' \circ h(f)$ over D_2^n.

Since $abs, conc$ are monotonic, by definition,

$$\begin{aligned} h' \circ h(f)(z) &= Min(\{y \in D_2^n \mid h(f)(y) \cap RC(z) \neq \phi\}) \\ &= Min(\{y \in D_2^n \mid Max \circ abs \circ f \circ conc \circ LC(y) \cap RC(z) \neq \phi\}) \\ &= Min(\{y \in D_2^n \mid Max \circ abs \circ f \circ conc(y) \cap RC(z) \neq \phi\}) \end{aligned}$$

for $z \in D_2$. Since abs is centerized and by definition,

$$\begin{aligned} h''(f)(z) &= Min \circ abs(\{x \in D_1^n \mid f(x) \in conc \cdot RC(z)\}) \\ &= Min(\{abs(x) \mid f(x) \in conc \cdot RC(z)\}) \end{aligned}$$

Since h is necessary,

$$\begin{aligned} h''(f)(z) &= Min(\{abs(x) \mid f(x) \in conc \cdot RC(z)\}) \\ &= Min(\{abs(x) \mid abs \circ f(x) \in RC(z)\}) \\ &= Min(\{y \in D_2^n \mid \exists x \in conc(y) \text{ s.t. } abs \circ f(x) \in RC(x)\}) \end{aligned}$$

Thus, we conclude $h' \circ h(f) = h''(f)$. ■

Definition 3.17 *Let h_1 and h_2 be HOMTs. We denote $h_1 \preceq h_2$ if there exists a HOMT h_{21} such that $h_1 = h_{21} \circ h_2$, and $h_1 \simeq h_2$ (i.e., equivalent) if $h_1 \preceq h_2 \ \wedge \ h_2 \preceq h_1$.*

Example 3.18 *The equivalence of FSA and SIA (see Example 3.6 and 3.7) is proved by the existence of forward/backward conversion operators. These operators are easily found from Theorem 3.16. That is, the underlined U-HOMTs transform* FSA *to* SIA

$$\begin{aligned} & \underline{((id, id), -, (\sqsubseteq_{-0}, RC), Min)} \circ ((abs_b, abs_b^{-1}), +, (\sqsubseteq_1, LC), Max) \\ = \ & \underline{((abs_b, abs_b^{-1}), -, (\sqsubseteq_{-0}, RC), Min)} \end{aligned}$$

and SIA *to* FSA

$$\begin{aligned} & \underline{((id, id), -, (\sqsubseteq_1, LC), Max)} \circ ((abs_b, abs_b^{-1}), -, (\sqsubseteq_{-0}, RC), Min) \\ = \ & \underline{((abs_b, abs_b^{-1}), +, (\sqsubseteq_1, LC), Max)}. \end{aligned}$$

Note that this example corresponds to the reversal in [HL94].

3.2 Computation path analysis

3.2.1 Computation path analysis (CPA)

For simplicity, we first consider functional programs over flat domains, and the abstraction abs_b, that is, the abstract domain is a lattice consisting of two elements $\{\delta^0, \delta^1\}$, where δ^0 and δ^1 correspond to the bottom $\perp$ and evaluated values, respectively.

Computation path analysis over flat domains

Computation Path Analysis (CPA) detects the property dependency parameter set (PDPS), which is a set of all possible demand propagation patterns of functions. For instance, the possible demand patterns of $if(x,y,z)$ are $\{x,y\}$ and $\{x,z\}$, and its PDPS is $\{\{x,y\},\{x,z\}\}$. From these interpretations for primitive functions, CPA detects the PDPS of recursively defined functions. For instance, the PDPS of the function foo, defined by

```
(defun foo (x y) (if (zerop x) 1 (1+ (foo (1- x) (foo y x)))))
```

is computed as $\{\{x\}\}$.

The CPA algorithm is formalized as a collection of deduction rules on a finite set of the algebraic expressions [Ono88]. The natural interpretation on primitive functions of CPA have the following set-like expressions.

$(if\ x\ y\ z)$	$\lambda xyz.\{\{x,y\},\{x,z\}\}$
$(+x\ y)$	$\lambda xy.\{\{x,y\}\}$
$const(x)$	$\lambda x.\{\phi\}$
$\Omega(x)$	$\lambda x.\phi$

Note that interpretations in CPA for a constant function $const(x)$ and the undefined function $\Omega(x)$ (which is an initial value in fixed-point computation) are different. $\lambda x.\{\phi\}$ means that no demands are required to get a result, whereas $\lambda x.\phi$ means that no demand-propagation patterns exist (i.e. no way to evaluate).

Internal expressions in the algorithm have algebraic expressions below, which consist of function symbols $+, *$ and constant symbols $\mathbf{0}$, $\mathbf{1}$, with an ordering $M \sqsubseteq N$ if and only if $M + N = N$. Here, $*$, and $+$, $\mathbf{0}$, and $\mathbf{1}$ correspond to the merge of paths, the collection of paths, ϕ, and $\{\phi\}$, respectively.

$(if\ x\ y\ z)$	$\lambda xyz.x * y + x * z$
$(+x\ y)$	$\lambda xy.x * y$
$const(x)$	$\lambda x.\mathbf{1}$
$\Omega(x)$	$\lambda x.\lambda x.\mathbf{0}$

The algebraic relations among these expressions for the CPA are as follows:

(redundancy elimination)	$x * x = x$	$x + x = x$
(commutativity)	$x * y = y * x$	$x + y = y + x$
(associativity)	$(x * y) * z = x * (y * z)$	$(x + y) + z = x + (y + z)$
(distributive laws)	$x * (y + z) = x * y + x * z$	
(zero element)	$\mathbf{0} * x = \mathbf{0}$	$\mathbf{0} + x = x$
(unit element)	$\mathbf{1} * x = x$	

Then, a recursive function, such as *Peyton-Jones*'s f (in pp.389 [PJ87])

```
(defun f (x y z p)
   (if (zerop p) (+ x z) (+ (f y 0 0 (1- p)) (f z z 0 (1- p)))))
```

is analyzed by the fix point computation as follows.

$$\begin{array}{lll}
(0) & \lambda xyzp.\mathbf{0} & \\
(1) & \lambda xyzp.p * x * z & \\
(2) & \lambda xyzp.p * x * z + p * y * z & \\
(3) & \lambda xyzp.p * x * z + p * y * z + p * z & \\
(4) & \lambda xyzp.p + p * x * z + p * y + p * y * z + p * z & \\
(5) & \lambda xyzp.p + p * x * z + p * y + p * y * z + p * z & \quad \textit{converged}
\end{array}$$

Thus, PDPS of f is $\{\{p\}, \{p, x, z\}, \{p, y\}, \{p, y, z\}, \{p, z\}\}$.

Computation path analysis with modes

When analyzing programs over recursively defined data structures such as lists, a two-point abstract domain $\{\delta^0, \delta^1\}$ is not enough. Such an abstract domain represents only whether demands are propagated to variables. However, recursively defined data structures have demand levels; that indicate how deeply data need to be evaluated. For instance, (*length* x) which returns the length of a list x requires the evaluation of x, but does not require the full structure of x. It requires only the evaluation of the spine of x (i.e., the *cdr* direction), and does not require leaves of x (i.e., the *car* parts). This is called *tail strictness* [WH87]. Similarly, (*sum* x) which returns the sum of a list x requires the full evaluation of x, and (*search*0 x), which returns *true* if 0 is found in x and returns *false* otherwise, requires the synchronous evaluation of leaves (the car part) to the evaluation of its spine (the cdr part). They are called *total strictness* and *head strictness*, respectively. (Head strictness is equivalent to H-strictness in [Bur90].)

Such depth of demands is expressed by introducing *modes*. A *mode* "$:\mu$" is an index to a variable x and denoted as $x^{:\mu}$. The requirements for a set of modes are that:

1. it be a finite complete lattice, and

2. the result of demand merging of modes $:\mu$ and $:\nu$ be in a mode $lub(:\mu, :\nu)$,

and the additional algebraic rule is

$$\text{(lub)} \quad x^{:\mu} * x^{:\nu} = x^{lub(:\mu,:\nu)}$$

which is applied for merging information of a multiply used variable x.

As a HOMT, $lub(:\mu, :\nu)$ corresponds to the interpretation of a `let` sentence. The set of modes corresponds to an abstract domain and

$$lub(:\mu, :\nu) = abs(\{x \sqcup y \mid x \in conc(:\mu),\ y \in conc(:\nu)\})$$

where $conc$ is abs^{-1}.

Example 3.19 *CPA with modes can be applied to a list structure, and to any recursive data structures as well. For instance, the simplest example of a recursive data structure is the domain of* lazy integers *(see Fig. 3.4). Let us consider the even/odd analysis, which*

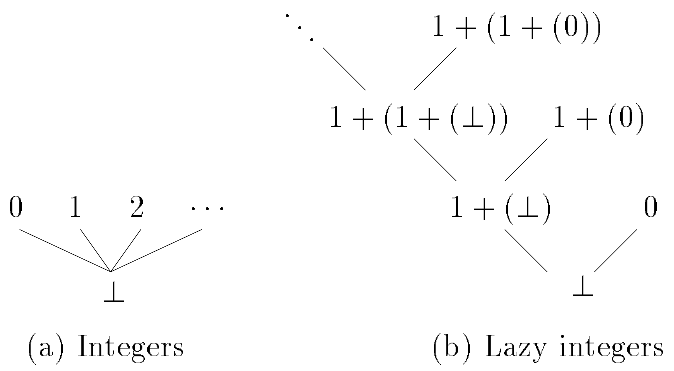


Figure 3.4: Two kinds of integer domains

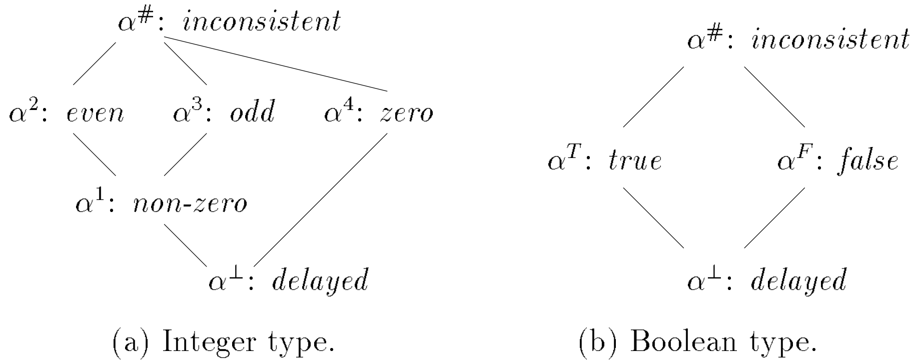


Figure 3.5: Abstract domains for even-odd analysis

detects what kinds of inputs are required when the output is even (or odd). The mode definition, which consists of the design of abstract domains and abstract interpretation of primitive functions, is given in Fig. 3.5 and 3.6. Then, Fig. 3.7 shows the (part of) analysis result for

```
(defun lazy-add (x y) (if (zerop x) y (1+ (lazy-add (1- x) y))))
```

when the output is odd.

3.2.2 CPA as HOMT

To describe and/or implement the CPA algorithm, the formalization by modes is useful. However, to compare CPA with various analyses, the formalization by the quadruplet representation, HOMT, is convenient. First, we will formalize CPA over flat domains as necessary HOMTs, and later by adding the treatment of modes, we will formalize CPA over nonflat domains.

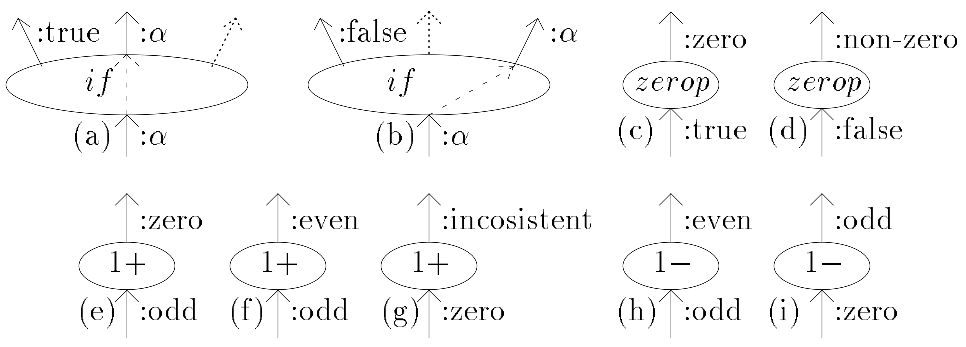


Figure 3.6: Inference for primitive functions for even-odd analysis

CPA over flat domains as a HOMT

CPA cannot be treated as a single U-HOMT, but represented as a composition of two U-HOMTs. We will see this with the examples, $if(x,y,z)$ and $serial\text{-}or(x,y,z)$, where $serial\text{-}or(x,y,z)$ returns values $true$ when $(x,y,z) = (true,\bot,\bot)$, $(false,true,\bot)$, $(false,false,true)$, and returns $false$ only when $(false,false,false)$. By definition, $if_{CPA} = \lambda xyz.\{\{x,y\},\{x,z\}\}$ and $serial\text{-}or_{CPA} = \lambda xyz.\{\{x\},\{x,y\},\{x,y,z\}\}$ [Ono88].

The first step to get a HOMT h_{CPA} is the construction of the function inverse. This U-HOMT h_1 has a quadruplet representation $((id,id),-,(\sqsubseteq_{-0},RC),Min)$ such as

$$\begin{array}{lll}
h_1(if) : & \{5\} \rightarrow \{(true,5,\bot),\ (false,\bot,5)\}, \\
& \{4\} \rightarrow \{(true,4,\bot),\ (false,\bot,4)\}, \\
& \cdots. \\
h_1(serial\text{-}or) : & \{true\} \rightarrow \{(true,\bot,\bot),\ (false,true,\bot),\ (false,false,true)\}, \\
& \{false\} \rightarrow \{(false,false,false)\}. \\
h_1(\Omega) : & \{\mathsf{any}(\neq\bot)\} \rightarrow \phi.
\end{array}$$

and the initial value of the algorithm for an n-ary function is $\lambda x_1 \cdots x_n.\phi$. The second step abstracts differences of values, but keeps the difference between an evaluated value and $\bot$. This U-HOMT h_2 has a quadruplet representation $((abs_b, abs_b^{-1}),+,(\subseteq,id),id)$ such as

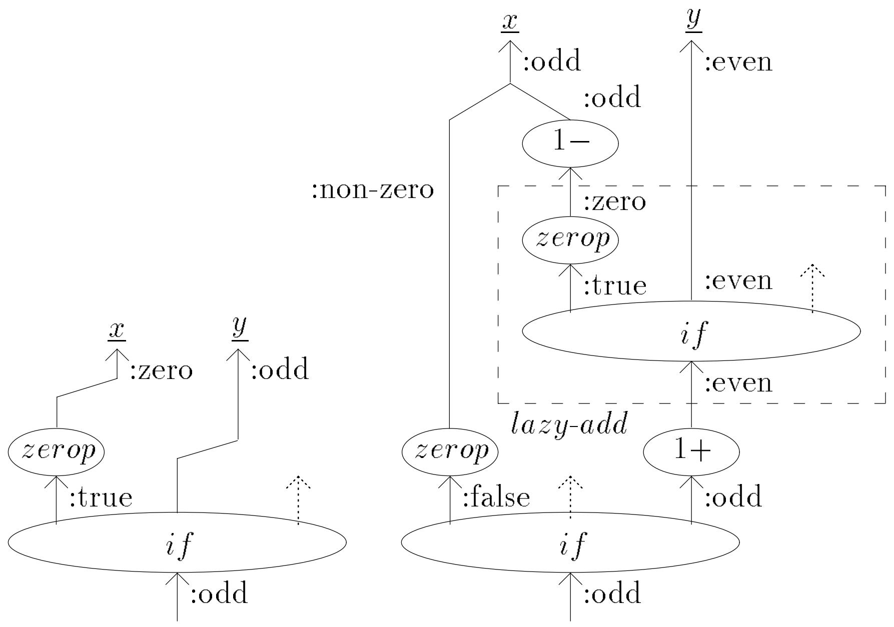
$$\begin{array}{ll}
h_2 \circ h_1(if) & : \{\mathbf{1}\} \rightarrow \{(\mathbf{1},\mathbf{1},\mathbf{0}),(\mathbf{1},\mathbf{0},\mathbf{1})\}. \\
h_2 \circ h_1(serial\text{-}or) & : \{\mathbf{1}\} \rightarrow \{(\mathbf{1},\mathbf{0},\mathbf{0}),(\mathbf{1},\mathbf{1},\mathbf{0}),(\mathbf{1},\mathbf{1},\mathbf{1})\}. \\
h_2 \circ h_1(\Omega) & : \{\mathbf{1}\} \rightarrow \phi
\end{array}$$

which exactly correspond to if_{CPA} and $serial\text{-}or_{CPA}$. Thus, CPA as a HOMT has a quadruplet representation

$$((abs_b, abs_b^{-1}),+,(\subseteq,id),id) \circ ((id,id),-,(\sqsubseteq_{-0},RC),Min).$$

CPA over nonflat domains as a HOMT

Next we consider CPA on lists as a HOMT. Apart from CPA over flat domains, we divide abs_b to abs_{int} and abs_{Bool}, which are corresponding to integers and Booleans, respectively.



(a) Initial condition (b) Inductive step

Figure 3.7: Analysis on $lazy\text{-}add(x, y)$ $: odd$

abs_{int} is defined identically to abs_b on integers, and abs_{Bool} is defined to be an identity map on Booleans. That is, an abstraction on Boolean values is not necessary because the Boolean domain is originally finite, and a naive abstraction on Boolean values loses the information on dataflow at conditional branches, which is crucial in analyses on nonflat domains.

The following domain abstraction $abs_{list(int)}$ ignores differences among values, but keeps shapes of list data structures.

Definition 3.20 *The base domain abstraction $abs_{int}: int \to \{\mathbf{0}, \mathbf{1}\}$ on a flat domain int is extended inductively according to the structure of domain D. For instance, the extension to list(int) is*

$$abs_{list(int)} : x \to \begin{cases} constructor(abs_{int}(y), abs_{list(int)}(z)) & if \quad x = constructor(y, z) \\ \mathbf{NIL} & if \quad x = Nil \end{cases}$$

$abs_{list(int)}(list(int))$ is still an infinite domain, thus we need further abstraction. Typical properties of interest are head/total/tail strictness, which were firstly proposed by projection analysis [WH87, DW90]. The abstraction for CPA for head/total/tail strictness are given below.

Example 3.21 *Let the abstract domain consisting of η's be as shown in Fig. 3.8, and let abs_{all} be the abstraction*

$$abs_{all} \ : \ list(int) \to \{\eta^0, \eta^1, \eta^2, \eta^3, \eta^4, \eta^5, \eta^6, \eta^7, \eta^8, \eta^{nil}\}.$$

The meaning of η's are

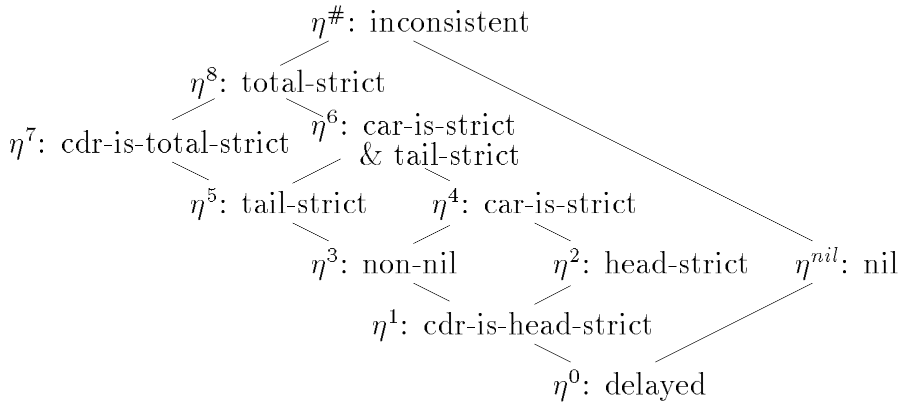


Figure 3.8: Abstract domain for strictness over a list domain

η^0 and η^{NIL} consists of a single element $\perp$ and NIL, respectively.
η^1 collects the elements in that their cdr parts are head-strict.
η^2 collects the elements that are head-strict.
η^3 collects the elements that are non-nil lists, but neither their head-parts nor their spines are evaluated.
η^4 collects the elements in that their head-parts are evaluated, but spines are not evaluated.
η^5 collects the elements in that their spines are evaluated, but head-parts and else remain unevaluated.
η^6 collects the elements in that their head-parts and spines are evaluated.
η^7 collects the elements in that every parts except head-parts are evaluated.
η^8 consists of the elements that are completely evaluated.

Then the quadruplet representation of CPA for head/total/tail strictness is

$$((abs_{all}, abs_{all}^{-1}), +, (\subseteq, id), id) \ \circ \ ((id, id), -, (\sqsubseteq_{-0}, RC), Min).$$

Remark 3.22 *Compare η's in Fig. 3.8 with the 6-point abstract domain in [Bur90]. The difference is the addition of η^1 and η^2, which enables us to detect head strictness by CPA. Note that the lattice in Fig. 3.8 does not work correctly with (mere) strictness analyses.*

3.2.3 Comparison with various analyses

The hierarchy of static analyses arises for two reasons :

- The objective property of a program itself is more informative on SA_2 than SA_1. (Property hierarchy)
- The objective property of a program is the same for SA_1 and SA_2, but the abstraction of SA_2 is more detailed than that of SA_1. (Approximation hierarchy)

In the following, we will investigate these hierarchies.

Property hierarchy among strictness analyses and CPA

An example of a property hierarchy appears already in Section 3.1.2 as forward/backward conversion among SIA and FSA (see Example 3.6 and 3.7). They are equivalent, and the abstract domain contraction leads to BSA from SIA. Another property hierarchy is found in the relation among CPA, SIA, and relevance analysis [Ono88], where a relevance analysis detects a set of parameters that *may* be evaluated. For example, *Peyton-Jones*'s f (in pp.389 [PJ87], also pp.54 in this thesis) is analyzed as

$$\begin{cases} \text{PDPS} & \{\{p\},\{p,x,z\},\{p,y\},\{p,y,z\},\{p,z\}\} \\ \text{relevant parameters} & \{x,y,z,p\} \\ \text{requisite parameters} & \{p\} \end{cases}$$

Roughly speaking, the union of all elements in PDPS is a set of relevant parameters, and the intersection is a set of requisite parameters. Therefore, CPA is more powerful than both strictness analysis and relevance analysis.

Example 3.23 *From Theorem 3.16, SIA is induced from CPA as*

$$\begin{aligned} & ((id,id),+,(\sqsubseteq_{-0},RC),Min)\circ h_{CPA} \\ = \; & ((id,id),+,(\sqsubseteq_{-0},RC),Min)\circ((abs_b,abs_b^{-1}),+,(\subseteq,id),id) \\ & \qquad\qquad\qquad \circ\;((id,id),-,(\sqsubseteq_{-0},RC),Min) \\ = \; & ((abs_b,abs_b^{-1}),+,(\sqsubseteq_{-0},RC),Min)\circ((id,id),-,(\sqsubseteq_{-0},RC),Min) \\ = \; & ((abs_b,abs_b^{-1}),-,(\sqsubseteq_{-0},RC),Min) \\ = \; & h_{SIA} \end{aligned}$$

Similarly relevance analysis with a quadruplet representation $((abs_b,abs_b^{-1}),-,(\sqsubseteq_1,LC),Max)$ *is induced from CPA as*

$$\begin{aligned} & ((id,id),+,(\sqsubseteq_1,LC),Max)\circ h_{CPA} \\ = \; & ((id,id),+,(\sqsubseteq_1,LC),Max)\circ((abs_b,abs_b^{-1}),+,(\subseteq,id),id) \\ & \qquad\qquad\qquad \circ\;((id,id),-,(\sqsubseteq_{-0},RC),Min) \\ = \; & ((abs_b,abs_b^{-1}),+,(\sqsubseteq_1,LC),Max)\circ((id,id),-,(\sqsubseteq_{-0},RC),Min) \\ = \; & ((abs_b,abs_b^{-1}),-,(\sqsubseteq_1,LC),Max) \\ = \; & h_{RA} \end{aligned}$$

For simplicity, here we consider analyses only on flat domains, but the comparison is easily extended to nonflat domains. For instance, CPA for total/tail strictness (see Fig. 3.9) and SIA on 6-points domain [Bur90] has the similar property hierarchy, by the same discussion above except for replacing abs_b with $abs_\alpha(= abs_{\eta\to\alpha}\cdot abs_\eta)$.

Approximation hierarchy among CPAs

The approximation hierarchy arises from the fact that an analysis is a compile-time technique, whereas the objective property is a run-time property. Thus, approximation accuracy is traded off with computational feasibility (including termination). Therefore, even for the same objective property, there are many selections for approximation levels. These levels can be measured by domain abstractions.

The example is the relation among CPAs on non-flat domains (e.g. lists), which detect head/tail/total strictness. Recall CPA on lists in Example 3.21 with the quadruplet representation $h_{all} = ((abs_{all}, abs_{all}^{-1}), +, (\subseteq, id), id) \circ ((id, id), -, (\sqsubseteq_{-0}, RC), Min)$. Thus, their approximation hierarchy (shown in Fig. 3.9) follows from the reduction of quadruplet representations below. Note that abs_{η}, abs_{ϵ}, and $abs_{\eta\to\epsilon}$ are nonmonotonic.

$$\begin{array}{lcl}
h_{head} & = & ((abs_{\eta\to\epsilon}, abs_{\eta\to\epsilon}^{-1}), +, (\subseteq, id), id) \circ h_{all} \\
h_{tail/total} & = & ((abs_{\eta\to\alpha}, abs_{\eta\to\alpha}^{-1}), +, (\subseteq, id), id) \circ h_{all} \\
h_{total} & = & ((abs_{\alpha\to\beta}, abs_{\alpha\to\beta}^{-1}), +, (\subseteq, id), id) \circ h_{tail/total} \\
h_{tail} & = & ((abs_{\alpha\to\gamma}, abs_{\alpha\to\gamma}^{-1}), +, (\subseteq, id), id) \circ h_{tail/total} \\
h_{flat} & = & ((abs_{\beta\to\delta}, abs_{\beta\to\delta}^{-1}), +, (\subseteq, id), id) \circ h_{total} \\
 & = & ((abs_{\gamma\to\delta}, abs_{\gamma\to\delta}^{-1}), +, (\subseteq, id), id) \circ h_{tail} \\
 & = & ((abs_{\epsilon\to\delta}, abs_{\epsilon\to\delta}^{-1}), +, (\subseteq, id), id) \circ h_{head}
\end{array}$$

where

$$abs_{\eta\to\alpha} : \begin{cases} \eta^{\#} & \to \alpha^{\#} \\ \eta^{NIL} & \to \alpha^{nil} \\ \eta^{8} & \to \alpha^{6} \\ \eta^{7} & \to \alpha^{7} \\ \eta^{6} & \to \alpha^{4} \\ \eta^{5} & \to \alpha^{3} \\ \eta^{4}, \eta^{2} & \to \alpha^{2} \\ \eta^{3}, \eta^{1} & \to \alpha^{1} \\ \eta^{0} & \to \alpha^{0} \end{cases}
\qquad
abs_{\eta\to\epsilon} : \begin{cases} \eta^{\#} & \to \epsilon^{\#} \\ \eta^{nil} & \to \epsilon^{nil} \\ \eta^{6}, \eta^{5}, \eta^{4}, \eta^{3} & \to \epsilon^{3} \\ \eta^{8}, \eta^{2} & \to \epsilon^{2} \\ \eta^{7}, \eta^{1} & \to \epsilon^{1} \\ \eta^{0} & \to \epsilon^{0} \end{cases}$$

$$abs_{\eta\to\epsilon} : \begin{cases} \epsilon^{nil}, \epsilon^{3}, \epsilon^{2}, \epsilon^{1} & \to \delta^{1} \\ \epsilon^{0} & \to \delta^{0} \end{cases}$$

$$abs_{\alpha\to\beta} : \begin{cases} \alpha^{\#} & \to \beta^{\#} \\ \alpha^{nil} & \to \beta^{nil} \\ \alpha^{6} & \to \beta^{4} \\ \alpha^{5} & \to \beta^{3} \\ \alpha^{4}, \alpha^{2} & \to \beta^{2} \\ \alpha^{3}, \alpha^{1} & \to \beta^{1} \\ \alpha^{0} & \to \beta^{0} \end{cases}
\qquad
abs_{\alpha\to\gamma} : \begin{cases} \alpha^{\#} & \to \gamma^{\#} \\ \alpha^{nil} & \to \gamma^{nil} \\ \alpha^{6}, \alpha^{5}, \alpha^{4}, \alpha^{3} & \to \gamma^{2} \\ \alpha^{2}, \alpha^{1} & \to \gamma^{1} \\ \alpha^{0} & \to \gamma^{0} \end{cases}$$

$$abs_{\beta\to\delta} : \begin{cases} \beta^{nil}, \beta^{4}, \beta^{3}, \beta^{2}, \beta^{1} & \to \delta^{1} \\ \beta^{0} & \to \delta^{0} \end{cases}
\qquad
abs_{\gamma\to\delta} : \begin{cases} \gamma^{nil}, \gamma^{2}, \gamma^{1} & \to \delta^{1} \\ \gamma^{0} & \to \delta^{0} \end{cases}$$

Comparison with projection analysis

Head/tail strictness detection was firstly proposed as a projection analysis, where a projection of a domain D is an idempotent mapping over an extended domain $D_{\lightning}$ with a

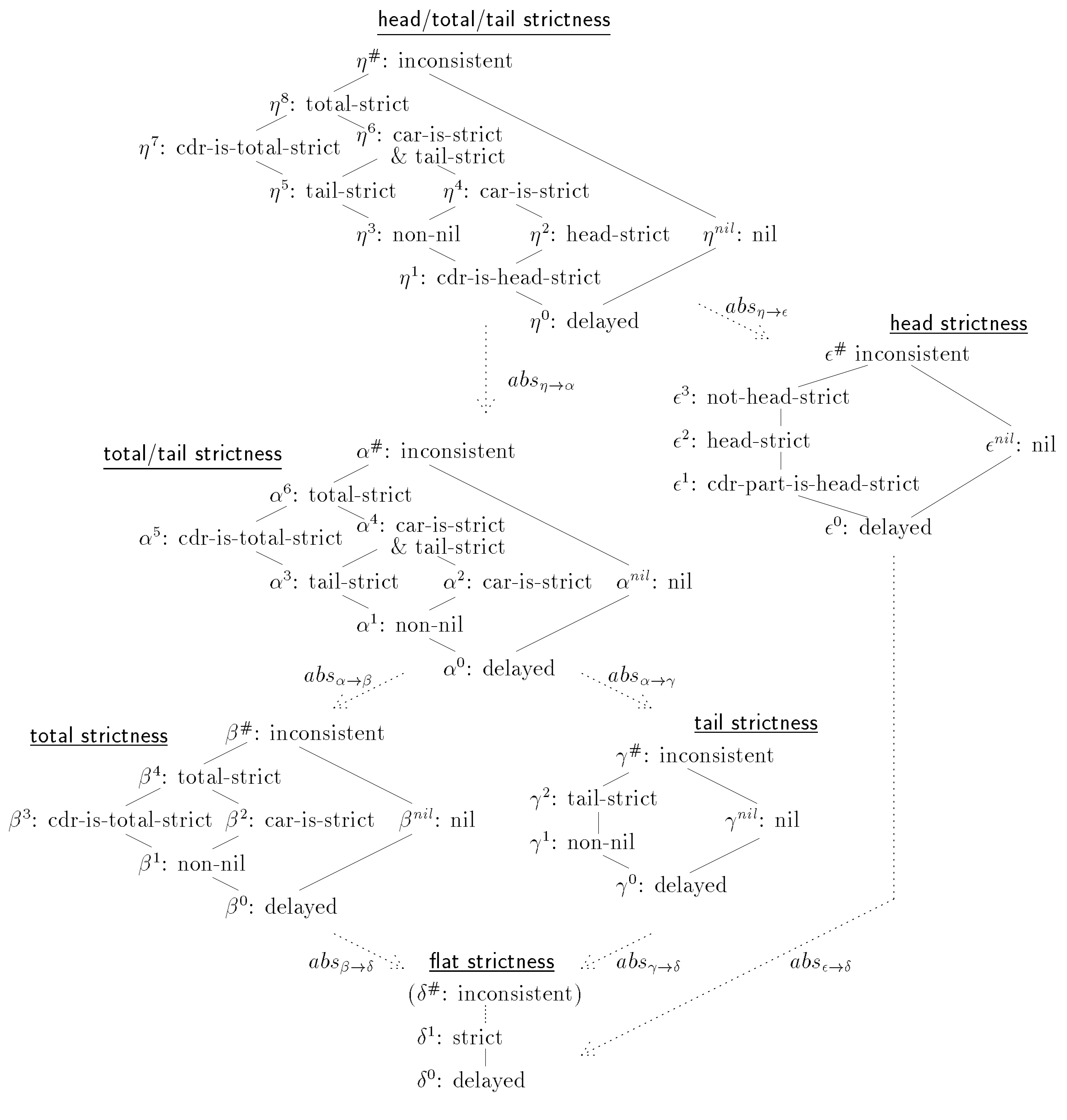
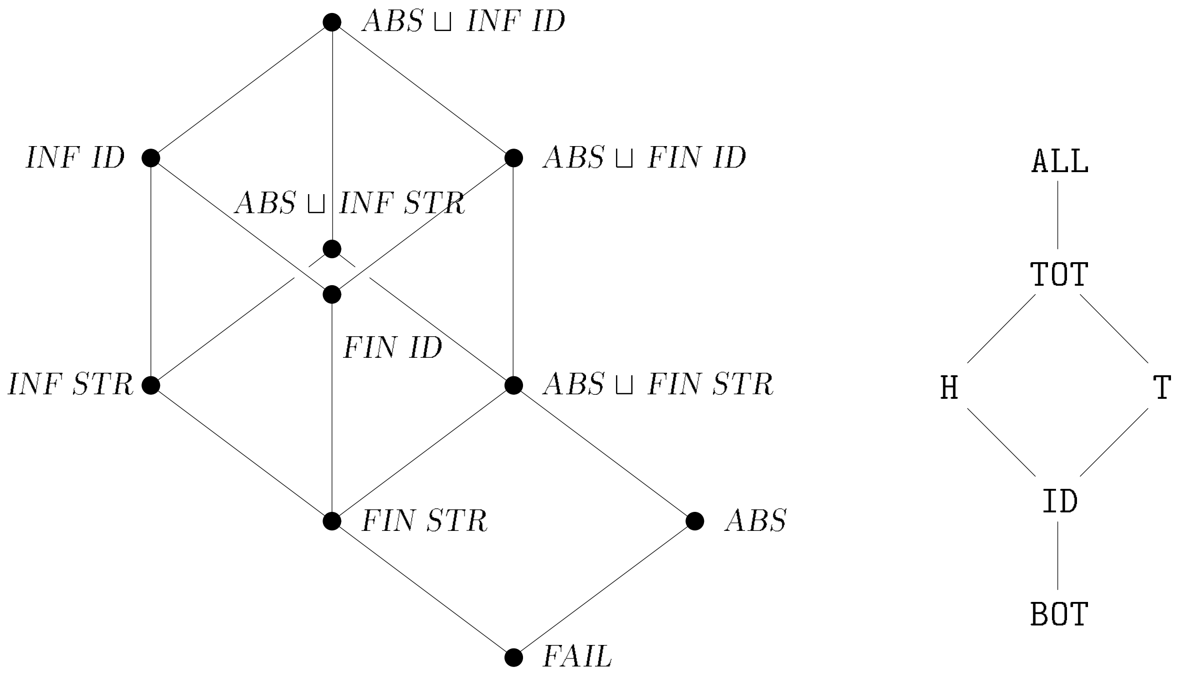


Figure 3.9: Hierarchy of abstract domains for strictness over lists



Lattice of projections for lists
(backward analysis)

Lattice of PERs for lists
(forward analysis)

Figure 3.10: Lattice of abstract domains for strictness of lists ([WH87], [Hun91])

special constant ↯ [WH87]. This analysis is incomplete in terms of its non-relational (or low-fidelity [DW91], i.e., the strict parameter of $if(x, y, y)$ is detected as x only) nature. It is enhanced to a relational (or high-fidelity) analysis in [DW90].

Burn tried to clarify the relationship between projection analysis and abstract interpretation, and obtained only tail-strictness and the very weak part (H_B analysis) of head-strictness detections [Bur90]. Ernoult and Mycroft also showed that a uniform abstract interpretation cannot treat head-strictness [EM91]. Their results show that the simple abstract interpretation on first-order abstract domains cannot treat head-strictness.

Hunt introduced a forward abstract interpretation over a partial equivalence relation (PER), and showed that the abstract domain in Fig. 3.10 can treat head-strictness [Hun91]. His basic idea is that a PER expresses the range of values of an input variable that does not affect the output. For instance, the irrelevance of a variable in the input is expressed such that any substitution to the variable does not affect the value of the output.

I conclude that the head strictness detection, which requires simultaneous analyses of strictness and irrelevance, requires either higher-order abstract domains (as PERs in [Hun91], or the composition of HOMTs.

The table below briefly summarizes the correspondence between the projections analysis and CPA. For details, please see [小96].

	projection analysis	CPA
abstract domain	projection α	$\alpha(D_{\lightning}) - \{\lightning\}$
demand combination	$\sqcup$	$+$
demand merge	&	*lub*

projection	mode of CPA
FAIL	ϕ
ABS	**0**
STR	**S** (=**1**)
ID	**D** (={**0**, **1**})

3.3 Applications of computation path analysis

Analyses are the heart of compile-time optimizations. There are many applications of static analyses. For instance, the call-by-need to call-by-value transformation is a natural optimization based on strictness analyses [AH87, Bur91]. A binding time analysis is equivalent to a strictness analysis [Lau91], thus strictness analyses can also be applied for the control of partial evaluation. CPA is more powerful than strictness analyses, thus CPA can be applied for them as well (or better) [OTA86, 小87, OTA84].

In this section, the application of CPA other than listed above is presented. This is an error detection, called *anomaly detection* [小88]. For instance, consider functions

```
(defun easy (x y) (if (zerop 0) 1 (easy (1- x) (easy y x))))
(defun diverged (x y)
   (if (< (+ x y) 0) (diverged (1+ x) y) (diverged (1- x) y)
```

By CPA, the PDPS of `(easy x y)` and `diverged(x,y)` are $\{\{x\}\}$ and { } (empty), respectively. This means that the parameter y in `(easy x y)` is irrelevant and the function `diverged` diverges. Strictly speaking, such irrelevance and divergence may not be errors; for instance, consider that divergence of some expression does not affect the output of the function. Thus, instead of errors, we call them *anomalies*. These anomalies require more sophisticated observations than previously known ones, such as undefined and unreferred objects [AU77, JA84].

We will show that such anomaly detection not only works for external anomalies (as above), but also for more detailed internal anomalies, which will specify the source of errors in expressions. For this purpose, we will show how dataflow information with function bodies are reconstructed from PDPSs of functions. For simplicity, we use CPA on flat domains even for lists, but the extension to more sophisticated CPA is straightforward.

3.3.1 Intra-functional computation path analysis (VDPS analysis)

The computation path of the expression *exp* in a function body can be represented by a tuple of variables (including local variables), by performing the labeling transformation, which labels each output of each function in the function body, beforehand. This tuple is called the minimally sufficient value set (MSVS). The VDPS is the set of all MSVSs and is denoted by $\mathcal{V}(exp)$.

The VDPS analysis determines the VDPS of each expression in the function body based on the PDPS of each function. Consider, for example, the function

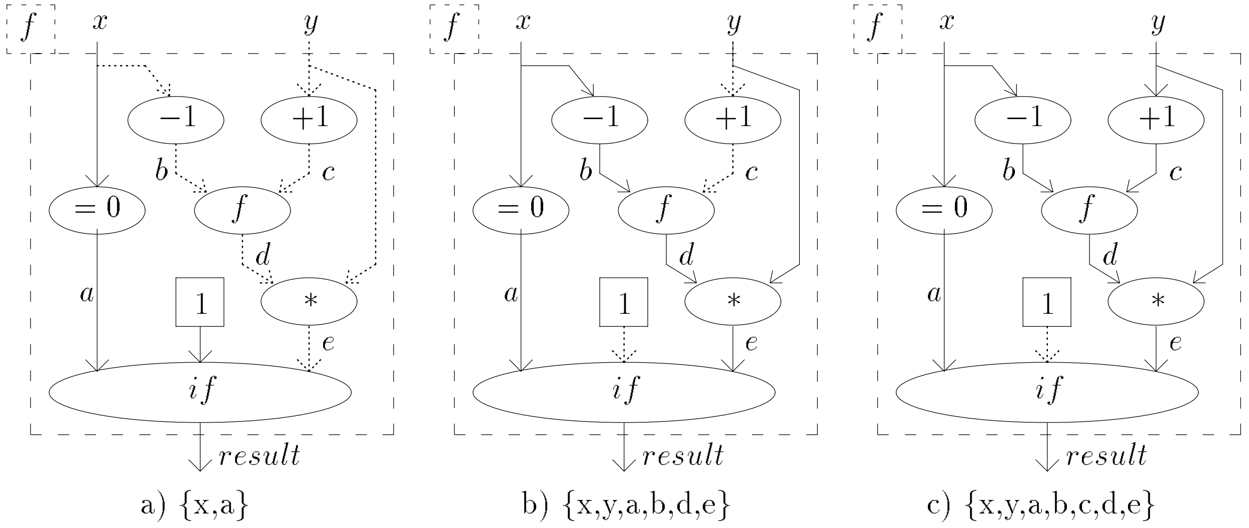


Figure 3.11: Relation between MSVSs of function $f(x, y)$ and its computation paths.

```
(defun f (x y) (if (zerop x) 1 (* (f (1- x) (1+ y)) y)))
```

Then, its PDPS is $\{\{x\}, \{x, y\}\}$.

By the labeling transformation, which assigns a fresh variable name (called a label variable) for each output of each function in the function body and flattens to the `letrec` expression [Hen80], we obtain the following function body of f:

```
(defun f (x y) (letrec ((a (zerop x)) (b (1- x)) (c (1+ y)) (d (f b c))
                        (e (* d y)) (result (if a 1 e)))
                  result))
```

Let $\otimes$ be an operator on a power set such that $A \otimes B = \{a \cup b \mid a \in A, b \in B\}$. Then VDPSs are computed by the recursive procedure that

$$\mathcal{V}(z) \;=\; \cup_{(x_{i_1},\cdots,x_{i_m}) \in PDPS(g)}\; \mathcal{V}(x_{i_1}) \otimes \cdots \otimes \mathcal{V}(x_{i_m}) \otimes \{(x_{i_1}, \cdots, x_{i_m})\}$$

where z is locally defined as $z = g(x_{i_1}, \cdots, x_{i_m})$. Thus $\mathcal{V}(result)$ of the output $result$ of function f is given by

$$\{\{x, a\}, \{x, y, a, b, d, e\}, \{x, y, a, b, c, d, e\}\}.$$

Each MSVS corresponds, as in Fig. 3.11, to the computation path leading to the local variable $result$ indicated by a line. The algorithm in the VDPS analysis corresponds to determining the transitive closure of the variable set on which each expression of the function body depends. For details, see [OTA86] and [小87].

3.3.2 Anomaly detection

An MSVS, which is an element of a VDPS, is a tuple of parameters, local variables, and/or label variables that are referred to in the computation path of the function. An internal anomaly, such as irrelevant local variables and diverged local variable definitions, can be detected in the same way as an external anomaly. The unique difference with an external anomaly is the presence of diverging loops in a function body that does not affect to the result of the function. Thus, additional loop detection and its propagation is required.

The diverging loop is determined essentially by determining self (circular) reference, i.e., whether the VDPS of the variable contains the same variable. Assume, for example, that the VDPS of a variable a is $\{\{b, c\}, \{a, b, c\}\}$. Then the latter MSVS $\{\{a, b, c\}\}$ contains variable a. This MSVS implies that the definition circulates so that the value of variable a is required in the computation of the value of a. Then it is regarded as a diverging loop.

In the loop detection for the functional language containing a delayed evaluation, the variable reference mode plays an important role in discriminating between the stream and the diverging loop. Consider, for example,

```
(letrec ((a (cons 1 a)) (b (cons-stream 1 b)) (c (car a)) .... )
```

Then, $\mathcal{V}(a) = \{\{a\}\}$, $\mathcal{V}(b) = \{\{b\}, \{\ \}\}$, and $\mathcal{V}(c) = \{\{a\}\}$. The computation paths for a and b are both in loops. The definition of value a circulates and diverges, while that of b gives a stream (1 1 1 ...). The difference can be detected from the fact that every MSVS of value a is with self reference, while some MSVS of value b is without self reference.

The divergence of a variable affects the divergence of the variable that refers to that variable. Consider, for example, the MSVS of the value c in the foregoing case. Only MSVS of c refers to the value of a, thus c also diverges. Consequently, by propagating the divergence based on the data dependency given as the result of the VDPS analysis.

To be precise, anomaly detection consists of two steps.

Step 1. Detection of an erroneous expression (diverging loop detection)

For a variable z, if each MSVS in $\mathcal{V}(z)$ contains z (i.e., each MSVS is with self reference), z is detected as an error.

Step 2. Propagation of errors to local variables.

Assume $z_1, z_2, \cdots$ be erroneous. For a variable z, if each MSVS in $\mathcal{V}(z)$ contains some z_i, then z is also detected as an error. If some MSVS in $\mathcal{V}(z)$ contains some z_i, then z is detected as a warning.

Examples of internal anomaly detection

An internal anomaly is detected based on the result of the VDPS analysis after applying the labeling transformation to the source program.

Example of program.

```
(defun foo (x y z)
   (letrec ((a (+ x y)) (b (cons y b)) (c (cons-stream y c))
            (d (if (> a 0) (car b) (car c))))
```

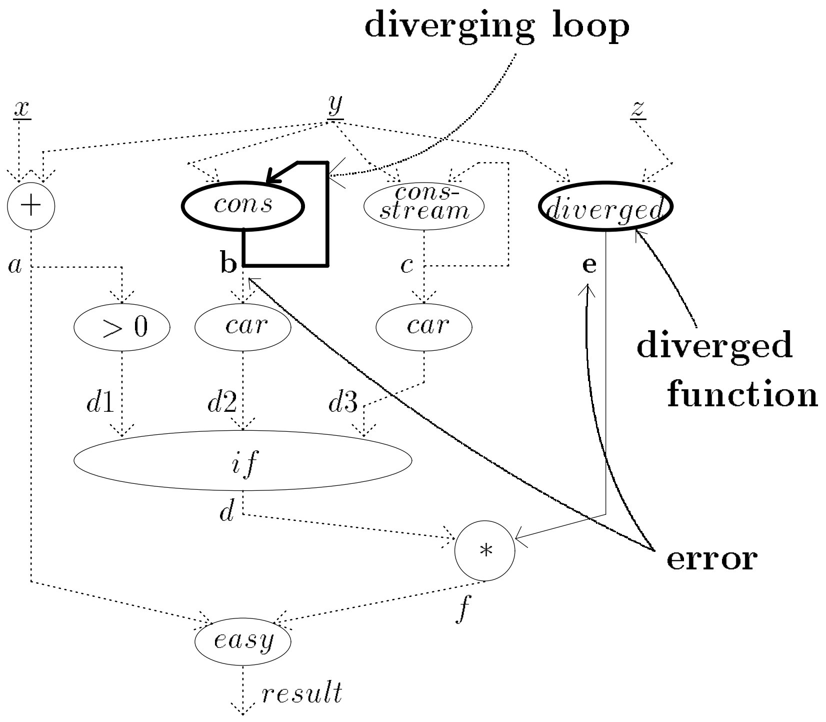


Figure 3.12: Diverging loop detection (Step 1)

```
        (easy a (* (diverged y z) d))))
```

Then PDPS of `foo` is $\{\{x, y\}\}$ (z is irrelevant), and after labeling transformation we obtain

```
  (defun foo (x y z)
     (letrec ((a (+ x y)) (b (cons y b)) (c (cons-stream y c))
             (d1 (> a 0)) (d2 (car b)) (d3 (car c))
             (d (if d1 d2 d3)) (e (diverged y z)) (f (* d e))
             (result (easy a f)))
         result))
```

After VDPS analysis, we obtain

$$\mathcal{V}(result) = \{\{x, y, a\}\},\ \mathcal{V}(a) = \{\{x, y\}\},\ \mathcal{V}(b) = \{\{y, b\}\},\ \mathcal{V}(c) = \{\{y, c\}, \{y\}\},$$
$$\mathcal{V}(d) = \{\{x, y, a, b, d1, d2\}, \{x, y, a, c, d1, d3\}, \{x, y, a, d1, d3\}\},$$
$$\mathcal{V}(d1) = \{\{a\}\},\ \mathcal{V}(d2) = \{\{b\}\},\ \mathcal{V}(d3) = \{\{c\}\},\ \mathcal{V}(e) = \phi,$$
$$\mathcal{V}(f) = \{\{x, y, a, b, d, d1, d2, e\}, \{x, y, a, c, d, d1, d3, e\}, \{x, y, a, d, d1, d3, e\}\}.$$

A variable z is detected to be irrelevant if each MSVS in $\mathcal{V}(result)$ does not contain z. Thus, in the function body of `foo`, variables $b, c, d, d1, d2, d3, e, f$ are detected as irrelevant.

Step 1. Detection of an erroneous expression (diverging loop detection)

In the case of function foo, for example, MSVS $\{y, b\} \in \mathcal{V}(b)$ of $b = cons(y, b)$ is $b \in \{y, b\}$, thus b is in a diverging loop. Since $\mathcal{V}(e)$ is empty, e is diverging. They correspond to the thick line in Fig. 3.12.

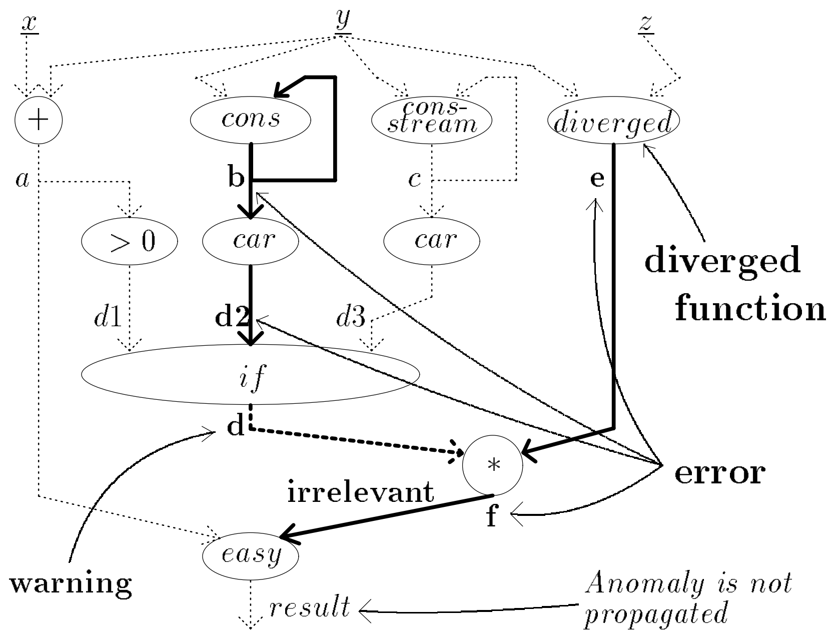


Figure 3.13: Error propagation to output and local variables (Step 2)

Step 2. Propagation of errors to local variables.

Step 2 detects which variables are affected by erroneous variables b and e. In Fig. 3.13, the part indicated by the thick line is the computation path that is the cause of an error, or the computation path that propagates errors. The part indicated by the thick dotted line is the computation path that propagates warnings.

From the result of VDPS analysis, all MSVS of $d2$ contain b and all MSVS of f contain e, thus errors are propagated to $d2$ and f. Similarly, some MSVS of d contains b, thus it is worth a warning message to d. No errors are reported for variables $result$, a, $d1$, $d3$, x, y. Note that $result$ is not affected by e, since the second parameter f of `easy` is irrelevant.

3.4 Implementation of automatic analyzer generator

3.4.1 Overview of automatic analyzer generator

Our experimental automatic analyzer generator runs on VAX/VMS. The system is implemented by Common Lisp, and its size is about 3K lines. The specifications of the analysis are given as a table, which is about 0.1K lines. This system also has a simple interactive user interface [小91].

An overview of the system is shown in Fig. 3.14. First, the user defines the mode definition of CPA by using the table. (Recall Fig. 3.5 and 3.6 for the even/odd analysis.) Then, the system automatically generates the CPA program by supplementing the tables in the specification and combining the general purpose least fix point calculator. Finally, the program to be analyzed is given to the generated CPA analyzer with specified modes.

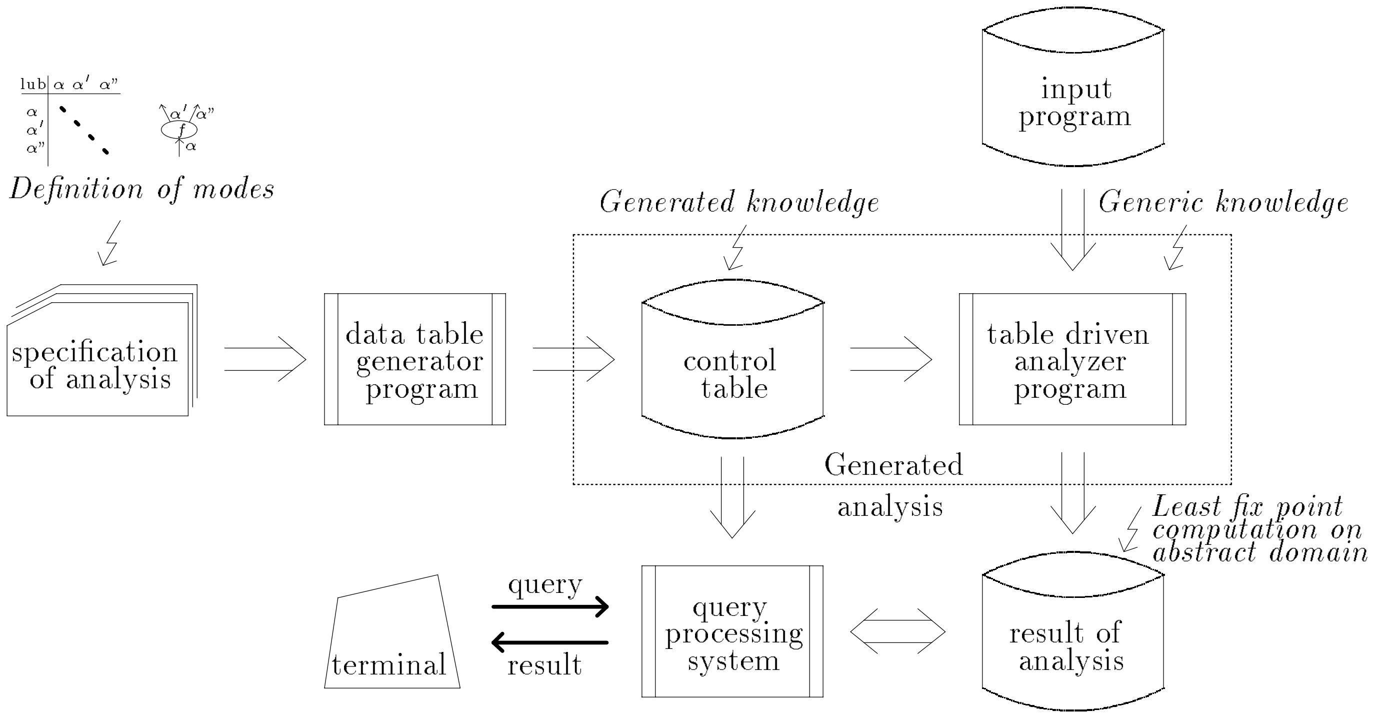


Figure 3.14: Analyses based on an automatic analyzer generator.

```
(defun lazy-add (x y) (if (zerop x) y (1+ (lazy-add (1- x) y))))
(defun lazy-mult (x y)
  (if (zerop x) (zero) (lazy-add y (lazy-mult (1- x) y))))
(defun evenp-1 (x)
  (if (zerop x) (true) (let ((y (1- x)))
                         (if (zerop y) (false) (evenp-1 (1- y))))))
(defun evenp-of-n*inc-n (n) (evenp-1 (lazy-mult n (1+ n))))
```

Figure 3.15: Examples of functions for even-odd analysis

```
 1. (load "primfunc-even-odd")          ;; loading CPA mode definition data
 2. (load "funcs-for-even-odd")         ;; loading functions to be analyzed
 3. (test-example-num :lub :fully-lazy-lub)     ;; generate analyzer
 4.
 5. (analyze '(lazy-add lazy-mult evenp-1 evenp-of-n*inc-n))
 6. ;Converged after 6 iterations. CPU Time:37.94 sec., Real Time:38.07 sec.
 7.
 8. (show-all-possible-combinations 'evenp-of-n*inc-n :true)
 9. (EVENP-OF-N*INC-N 1 ((1 :ZERO)) ((1 :ODD)) ((1 :EVEN)))
10. (show-all-possible-combinations 'evenp-of-n*inc-n :false)
11. (EVENP-OF-N*INC-N 1)
12. (show-all-possible-combinations 'lazy-add :odd)
13. (LAZY-ADD 2 ((1 :ZERO) (2 :ODD)) ((1 :EVEN) (2 :ODD))
                ((1 :ODD) (2 :EVEN)) ((1 :ODD) (2 :ZERO)))
```

Figure 3.16: Interactive session logging of the even-odd analysis

Fig. 3.16 shows the interactive session of the even/odd analysis of `evenp-of-n*inc-n`. Line 1 loads the specification of the even/odd analysis. Line 2 loads the program to be analyzed. Line 3 specifies the generator options, where :fully-lazy-lub means that the deduction table for the *lub* operations and the control table are generated in a demand-driven manner. (From our experience, the demand-driven generation is a realistic choice, since the eager one often consumes more than several minutes or hours. In the demand-driven generation, the table is generated during the analysis, but this does not practically affect to the execution time of the analysis.) Line 5 commands the actual analysis (i.e., the least-fixed-point computation). The time denoted is obtained by the compile code of VAX Common Lisp running on VAX3500.

Line 8 asks for the possible combinations of the inputs when the output is *true*. In line 9, `(1 :ZERO)` means that the first parameter is requested to be 0, `(1 :ODD)` means that the first parameter is requested to be a positive odd integer, and `(1 :EVEN)` means that the first parameter is requested to be a positive even integer. In contrast, line 10 asks for the possible combinations of the inputs when the output is *false*. The answer (in line 11) means that there are no possible computation paths for the *false* output. This means that the multiplication of n and $n+1$ is always even.

Several automatic dataflow analyzer generators have been proposed [OO90, YHI93]. Cecil detects program anomalies of procedural programs, such as wrong ordering of executions [OO90]. . Our method differs in that an inter-functional analyses is applied, whereas Cecil uses an intra-procedural dataflow analysis for imperative programs.

Set constraints [HJ90, AKW95, BGW93, GTT93] introduced different kinds of analyses, called set-based analyses [Hei93, Hei94], and an automatic analyzer generator, called BANE (Berkeley ANalysis Engine), was proposed based on set-based analyses [SFA00, AFFS98, FFSA98]. In general, solving set constraints, which is a core technique of set-based analyses, is *NEXPTIME*-complete. The restriction avoiding intersection, union,

```
(defun append-1 (x y) (if (null x) y (cons (car x) (append-1 (cdr x) y))))
(defun reverse-1 (x)
  (if (null x) (nil) (append-1 (reverse-1 (cdr x)) (cons (car x) (nil)))))
(defun search-0 (x)
  (if (null x) (false) (if (zerop (car x)) (true) (search-0 (cdr x)))))
(defun len-1 (x) (if (null x) (zero) (1+ (len-1 (cdr x)))))
(defun sum-1 (x) (if (null x) (zero) (+ (car x) (sum-1 (cdr x)))))
```

Figure 3.17: Definitions of example functions for CPA

and complement operations reduces it to cubic time [MR00], and further sophisticated implementation techniques enables BANE to analyze large-scaled programs. Many practical examples can be found at `http://http.cs.berkeley.edu/Research/Aiken/bane.html`.

3.4.2 Design scheme for abstract domains

The result of the even-odd analysis is the same regardless of the order of tracing computation paths and the lub operations of modes. That is, functions over an abstract domain satisfy $f(x \sqcup y) = f(x) \sqcup f(y)$. In this sense, such analyses are *continuous*. However, some analysis of inductive properties on more complex structures such as lists would be sensitive to such order. In this section, we explain such a *non-continuous* scheme. The example is a strictness analysis over lists, which detects tail strictness, total strictness, and head strictness [WH87, Bur90].

Lists on integers are described by the recursive equation

$$list(Int) = nil + Int \times list(Int).$$

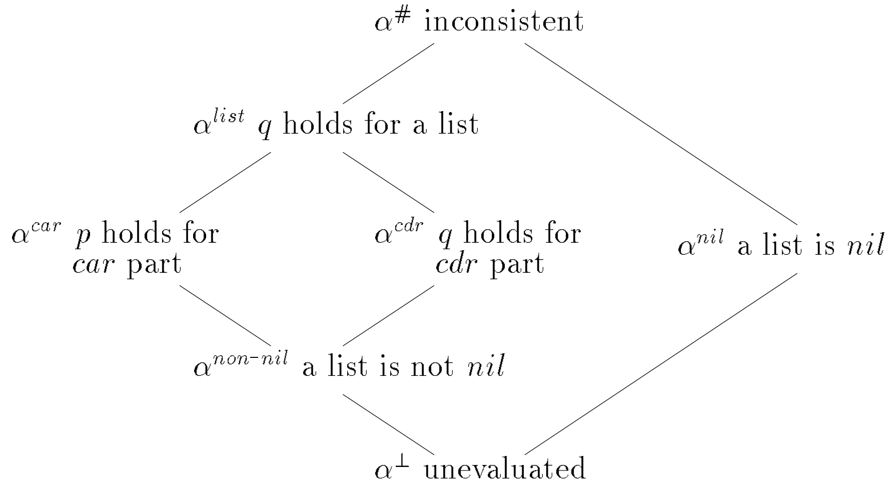
Corresponding to this expression, the above mentioned strictnesses are described as

tail strictness	$tail = nil + delay \times tail$
total strictness	$total = nil + strict \times total$
head strictness	$head = delay + strict \times head$

where *delay* means that the demand will not be propagated to the argument, and *strict* means that the demand will be propagated to the argument.

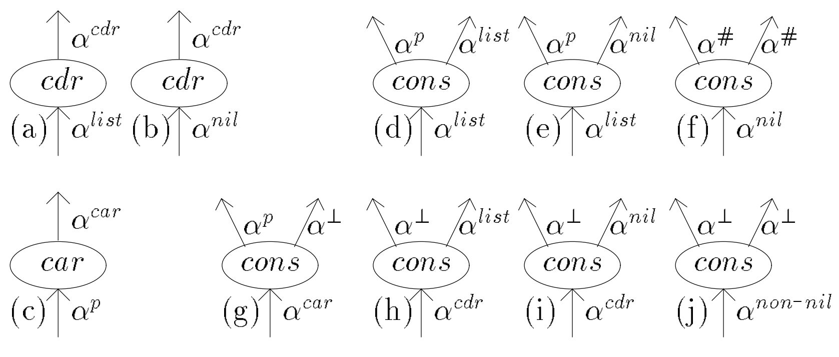
Intuitively, tail strictness expresses that the evaluation of the result requires the evaluation of the spine (top level structure) of an input list. Total strictness expresses that the evaluation of the result requires the evaluation of the whole structure of an input list. For instance, consider the functions in Fig. 3.17; $reverse\text{-}1(x)$, reverse an input list, $search\text{-}0(x)$, return $true$ if 0 is found and $false$ otherwise, $len\text{-}1(x)$, return the length of an input list, and $sum\text{-}1(x)$, return the sum of all elements. Then, the function $len\text{-}1(x)$ and $sum\text{-}0(x)$ are tail-strict and total-strict, respectively.

Head strictness expresses that the evaluation of the result requires the simultaneous evaluation of the element and the top level structure of an input list. For instance, the function $search\text{-}0(x)$ is head-strict.



$\alpha^{\#}$ inconsistent
α^{list} q holds for a list
α^{car} p holds for *car* part
α^{cdr} q holds for *cdr* part
α^{nil} a list is *nil*
$\alpha^{non\text{-}nil}$ a list is not *nil*
$\alpha^{\perp}$ unevaluated

(1) Design scheme for an abstract domain of lists



(a) cdr α^{list} α^{cdr}
(b) cdr α^{nil} α^{cdr}
(c) car α^{p} α^{car}
(d) cons α^{list} α^{p} α^{list}
(e) cons α^{list} α^{p} α^{nil}
(f) cons α^{nil} $\alpha^{\#}$ $\alpha^{\#}$
(g) cons α^{car} α^{p} $\alpha^{\perp}$
(h) cons α^{cdr} $\alpha^{\perp}$ α^{list}
(i) cons α^{cdr} $\alpha^{\perp}$ α^{nil}
(j) cons $\alpha^{non\text{-}nil}$ $\alpha^{\perp}$ $\alpha^{\perp}$

(2) Inference scheme for primitive functions on lists

Figure 3.18: Design scheme for an analysis on lists

Among these strictnesses, only tail strictness is continuous. Non-continuous properties are classified into monotonic and nonmonotonic ones. The following sections will explain that total strictness is monotonic and head strictness is nonmonotonic.

Design scheme for monadic properties: tail/total strictness analyses

The property P is monotonic if for each element x in an abstract domain satisfying P, any element $y(\neq \alpha^{\#})$ with $x \sqsubseteq y$ satisfies P. For an abstract domain of the monotonic property, functions are interpreted to satisfy that $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$.

For example, total strictness is monotonic. Recall that the abstract domains of total strictness were shown in Fig. 3.8, where β^2, β^3, and β^4 in that figure correspond to α^{car}, α^{cdr}, and α^{list} in Fig. 3.18, respectively.

Note that total strictness is not continuous; so the lub operation, which deduces α^{list} from α^{car} and α^{cdr} as an inductive step, must be done immediately after computation paths are merged. Otherwise, the information would be lost.

```
      (1) With  immediate lub operations
(show-all-possible-combinations 'reverse-1 :totally-strict)
(REVERSE-1 1 ((1 :NIL)) ((1 :TOTALLY-STRICT)))
(show-all-possible-combinations 'search-0 :false)
(SEARCH-0  1 ((1 :NIL)) ((1 :TOTALLY-STRICT)))
(show-all-possible-combinations '(len-1 sum-1) :int)
((LEN-1 1 ((1 :NIL)) ((1 :TAIL-STRICT)))
 (SUM-1 1 ((1 :NIL)) ((1 :TOTALLY-STRICT))))
      (2) Without immediate lub operations
(show-all-possible-combinations 'reverse-1 :totally-strict)
(REVERSE-1 1 ((1 :NIL)) ((1 :CAR-IS-STRICT :TAIL-STRICT)))
(show-all-possible-combinations 'search-0 :false)
(SEARCH-0  1 ((1 :NIL)) ((1 :CAR-IS-STRICT :TAIL-STRICT)))
(show-all-possible-combinations '(len-1 sum-1) :int)
((LEN-1 1 ((1 :NIL)) ((1 :TAIL-STRICT)))
 (SUM-1 1 ((1 :CAR-IS-STRICT :TAIL-STRICT)) ((1 :NIL))))
```

Figure 3.19: Total strictness analysis with/without immediate lub operations

Fig. 3.19 shows the difference between total strictness analysis *with* and *without* immediate lub operations.

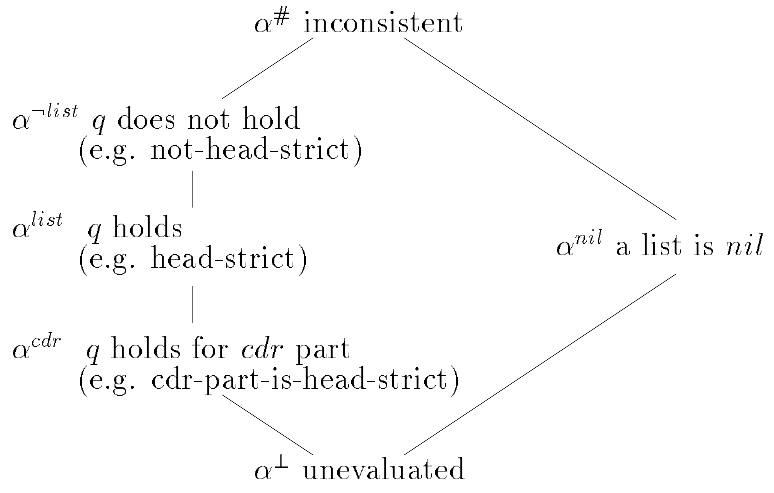
This situation occurs because at each function call non-nil lists are decomposed by *car* and *cdr*, and such list decompositions exceed the scope of the scheme without immediate lub operations. The immediate lub operation at each function call step restores the information and enables us to detect total strictness. Similarly, total strictness of a program that decomposes lists more than twice in its body cannot be detected. This also means that the unfolding transformation will be harmful to the analyses.

Note that tail strictness is correctly detected regardless of with/without immediate lub operations in Fig. 3.19. This is because tail strictness is essentially the property of the cdr part of a list.

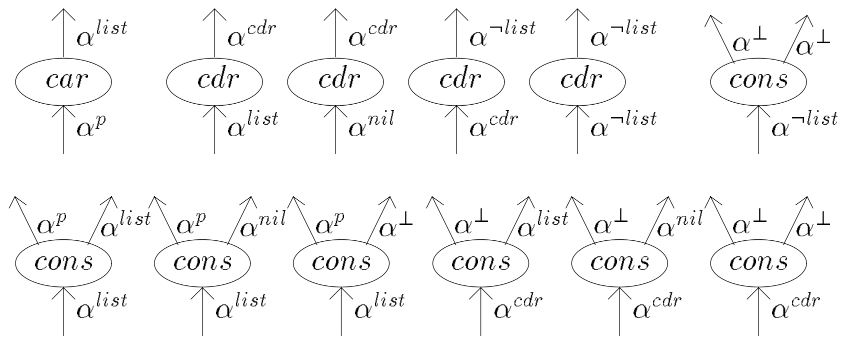
Design scheme with negative information: head strictness analysis

Head strictness contains a negative constraint so that it is nonmonotonic. Fig. 3.20 shows the scheme for the property with negative constraints. In Fig. 3.9, ϵ^2 expresses that the list is head strict, ϵ^1 that the cdr part is head strict, and ϵ^3 that the list is *not* head strict. They correspond to α^{list}, α^{cdr}, and $\alpha^{\neg list}$, respectively. As special cases, α^{cdr} contains the list in which its cdr part is undefined (i.e., $\perp$), and α^{list} contains the list in which its car part is strict (i.e., evaluated) and its cdr part is undefined (i.e., $\perp$). The nonmonotonicity appears in the deduction rules corresponding to the primitive function *cdr*: $\alpha^{list} \rightarrow \alpha^{cdr}$ and $\alpha^{cdr} \rightarrow \alpha^{\neg list}$, where $\alpha^{cdr} \sqsubseteq \alpha^{list} \sqsubseteq \alpha^{\neg list}$.

Fig. 3.21 shows the results of the head strictness analysis for *append*-0(x), *search*-0(x), *len*-1(x), and *sum*-1(x). From the result of total strictness analysis in the previous section, *search*-0(x) is detected as total strict when it returns *false*, and is detected as


$\alpha^{\#}$ inconsistent
$\alpha^{\neg list}$ q does not hold (e.g. not-head-strict)
α^{list} q holds (e.g. head-strict)
α^{cdr} q holds for *cdr* part (e.g. cdr-part-is-head-strict)
α^{nil} a list is *nil*
$\alpha^{\perp}$ unevaluated

(1) Design scheme for an abstract domain of lists


car α^{p} α^{list}
cdr α^{list} α^{cdr}
cdr α^{nil} α^{cdr}
cdr α^{cdr} $\alpha^{\neg list}$
cdr $\alpha^{\neg list}$ $\alpha^{\neg list}$
cons $\alpha^{\neg list}$ $\alpha^{\perp}$ $\alpha^{\perp}$
cons α^{list} α^{p} α^{list}
cons α^{list} α^{p} α^{nil}
cons α^{list} α^{p} $\alpha^{\perp}$
cons α^{cdr} $\alpha^{\perp}$ α^{list}
cons α^{cdr} $\alpha^{\perp}$ α^{nil}
cons α^{cdr} $\alpha^{\perp}$ $\alpha^{\perp}$

(2) Inference scheme for primitive functions on lists

Figure 3.20: Negation scheme for an analysis on lists

```
(show-all-possible-combinations 'search-0 :true :minimal nil)
(SEARCH-0 1 ((1 :HEAD-STRICT)))
(show-all-possible-combinations 'search-0 :false :minimal nil)
(SEARCH-0 1 ((1 :NIL)) ((1 :HEAD-STRICT)))
(show-all-possible-combinations '(len-1 sum-1) :int :minimal nil)
((LEN-1 1 ((1 :NIL)) ((1 :CDR-IS-HEAD-STRICT)) ((1 :ANY-LISTS)))
 (SUM-1 1 ((1 :HEAD-STRICT)) ((1 :NIL))))
```

Figure 3.21: Head strictness analysis (with lub)

head strict when it returns *true*.

Note that when head strictness and tail strictness are merged, we can conclude neither total strictness nor head strictness (i.e., $lub(\eta^2, \eta^5) = \eta^6 \neq \eta^2, \eta^8$ in Fig. 3.9). For instance, consider $find$-$0(x)$, which returns n if the first 0 is found at the n-th position in x. Then $find$-$0(x) + len$-$1(x)$ requires that the first occurrence of x is head-strict and the second occurrence of x is tail-strict, but if 0 is at the first position in x, neither total strictness nor head strictness holds. As illustrated by the example, the nonmonotonic property easily conflicts with other useful properties.

Chapter 4

Verification of binary properties based on abstract interpretation

Program errors cause failures during execution that can be classified into three categories.

1. Execution eventually stops as a result of illegal operations.

2. Execution does not terminate.

3. Execution results are not what was intended.

Errors of the first kind are detected by type inference, with such languages as ML and Haskell. In addition, although termination is in general undecidable, errors of the second kind can be automatically prevented by several techniques, such as simple termination [DJ90, DF85], termination analysis [Gis97], and dependency pairs [AG00].

The third kind of error cannot be prevented without a specification language, and there is always a trade-off between expressiveness and feasibility. If the aim is to express everything, it is easy to fall into the trap of undecidability. Moreover, too much expressiveness may make users hard to learn. For compile-time error detection, an automatic verifier that functions without any human guidance is desirable even if it verifies only partial specifications. Then the user can concentrate on what kind of properties, under the limitation of a simple and restricted specification language, properly approximate the program behavior.

By restricting both properties and languages, Le Métayer developed an automatic verification technique [LM95]. Its target language is a strongly-typed first-order functional language with product types and recursive types. The important restriction is that the conditional part of an if-expression contains only basic predicates (such as *null*, *leq*, *geq*, and *equal*) without any functional symbols.

He defines a language which prescribes a class of *uniform* predicates over recursive types. These predicates are constructed by predicate constructors from basic predicates on base types. As an example, his system expresses that a sort program returns a list of decreasing values (if the sort program terminates) and automatically verifies it. This property is called *orderedness* of the sort program, which is expressed by $\mathbf{true} \rightarrow \nabla geq(sort\ X)$ in his specification language. Note that the termination of the sort program is not verified; this verification is left to a termination analysis, such as discussed in Part 1.

Similar ideas are also found in [EM91, Jen94, Ben93, OO91a]; however, the significant differences are that

- binary predicates are allowed, and
- free variables in binary predicates are allowed.

The former extends the expressiveness of target properties from other flow analyses. The last maintains the power of inter-functional inferences. However, the expressive power of the specification language is still fairly restricted as a verification; for instance, the input-output relation cannot be described.

In this chapter, we reconstruct and extend the automatic verification technique of Le Métayer [LM95] based on a backward abstract interpretation [Oga99]. The abstract domain is designed as the set of predicates which are defined in our specification language and are constructed dependent to the program to verify. The termination and soundness proofs of the verification are naturally derived from the formalization as a backward abstract interpretation.

Similar to [LM95], we adopt the simple and inefficient sorting program for orderedness as a running example, but we also tried efficient sort programs, such as orderedness of *quick-sort* and *merge-sort* (both topdown and bottomup), and weak preservation of the *topdown merge-sort*. Recall that verification of these properties are quite messy even for human-being [Pau96].

Extensions are achieved by (1) using the input variable in function properties, (2) introducing new predicate constructors, and (3) using uninterpreted function/predicate symbols. They are demonstrated by verifying the sorting and formatting programs. The first and the second extensions expand the ability of the specification language so that it covers another major specification of the sorting program; namely, *weak preservation*, i.e., the input and the output are the same *set*. This is expressed by $\mathbf{true} \to \forall_l \exists_r equal \wedge \forall_r \exists_l equal(sort\ X)$. Note that since our specification language cannot express the number of elements in a list, our algorithm cannot detect the full specification of sort, called *preservation*, i.e., the input and the output are the same *multiset*.

The third extension expands the range of both target programs and the specification language. The expansion of target programs is achieved by loosening the restrictions on the conditional part of an if-expression. The running example is `format`, which formats a given sentence (expressed as a list of strings) to a specified width. The technique behind this extension is the use of *uninterpreted functions*. We also show how partial evaluation will cooperate with the verification. Other major specifications of `format` become expressible by the use of *uninterpreted predicates*. This technique drastically expands the expressive ability, such as the specification that the order of words is preserved by `format`.

At the end of this Chapter, a new termination criteria for a backward abstract interpretation over an infinite abstract domain is presented. This criteria is inspired by the fact that the constraints in terms of WQO guarantee the termination of symbolic model checking over infinite states [ACJYK96, AAB$^+$99, AN00, ACJYK00, FS98].

This chapter is organized as follows: Section 4.1 defines programming and specification languages. Section 4.2 provides the verification algorithm based on a backward

The language of expressions

$$\begin{array}{rcl}
E & = & BE \mid (BE, E) \mid (E, BE) \mid BE : E \mid f\ E \mid op\ E \mid (E) \mid \\
 & & \texttt{if}\ Cond\ \texttt{then}\ E_1\ \texttt{else}\ E_2 \mid \texttt{let}\ x = E_1\ \texttt{in}\ E_2\ \texttt{end} \mid \\
 & & \texttt{let}\ (x, y) = E_1\ \texttt{in}\ E_2\ \texttt{end} \mid \texttt{let}\ x : xs = E_1\ \texttt{in}\ E_2\ \texttt{end} \\
BE & = & x \mid c \\
Cond & = & p_u\ x \mid p_b\ (x, y)
\end{array}$$

$$\text{where} \left\{ \begin{array}{rlllll}
E \in & Exp & \text{expressions} & f \in & Fv & \text{function variables} \\
BE \in & BExp & \text{basic expressions} & x \in & Bv & \text{variables} \\
p_u, p_b \in & Pred & \text{basic predicates} & c \in & Const & \text{constants} \\
op \in & Prim & \text{primitive functions} & & &
\end{array} \right.$$

The syntax of programs $\quad Prog = \{\ \texttt{fun}\ f_i\ x_i = E_i\ ;\ \}$

The language of types

$$\begin{array}{rclcrcl}
T & = & T_G \mid T_F & & T_F & = & T_G \to T_G \\
T_G & = & T_U \mid T_P \mid T_R & & T_P & = & T_G \times T_G \\
T_R & = & \mu\alpha.\texttt{nil} + T_G : \alpha & & T_U & = & \tau \quad \text{(basic types)}
\end{array}$$

Figure 4.1: Syntax of Programming Language.

abstract interpretation. The termination and soundness proofs are also given. Section 4.3 demonstrates the verification of *orderedness* of the (simple but inefficient) sort program to explain the algorithm, and presents extensions and demonstrates the verification of major specifications of the sorting and formatting programs. Section 4.4 presents a new termination criteria in terms of a better-quasi-order (BQO).

4.1 Target programming and specification language

4.1.1 Programming language

The target language is a strongly-typed first-order functional language with Haskell-like syntax, in which product types and recursive types, such as lists $list(A) = \mu\alpha.nil + A \times list(\alpha)$, are allowed. We use : to mean infix *cons*, ++ to mean infix *append*, and [ ] to mean a list, namely, $[a_1, a_2, a_3] = a_1 : (a_2 : (a_3 : nil))$. The semantics of the language is given by an ordinary least fix-point computation. We assume that the language is strict, but the same technique can be applied to a lazy language as well. The precise syntax and semantics of the language are shown in Fig. 4.1 and Fig. 4.2. Parentheses in the syntax are used for either making pairs or clarifying the order of applications of infix operators. Basic concrete domains D_{Bool} and D_{Int} are flat CPO's (as usual), and the other concrete domains D_α of type α are constructed by the list and pair constructors. The interpretation ψ of expressions has the hidden argument, i.e., for simplicity the environment fve of function variables are omitted in Fig. 4.2.

$$
\begin{array}{lcl}
\varphi[\![\ \{\ \texttt{fun}\ f_i\ x_i = E_i\ ;\ \}\ ]\!] & = & fve \text{ whererec} \\
 & & \quad fve = [(\lambda y_1 \cdots y_n . if\ (bottom?\ y_1 \cdots y_n) \\
 & & \qquad then\ \perp\ else\ \psi[\![E_i]\!][y_j/x_j])/f_i] \\
\psi[\![C]\!]bve & = & \xi_c[\![C]\!] \\
\psi[\![x]\!]bve & = & bve[\![x]\!] \\
\psi[\![op\ E]\!]bve & = & \xi_f[\![op]\!](\psi[\![E]\!]bve) \\
\psi[\![p_u\ x]\!]bve & = & \xi_p[\![p_u]\!](bve[\![x]\!]) \\
\psi[\![p_b\ (x,y)]\!]bve & = & \xi_p[\![p_b]\!](bve[\![x]\!], bve[\![y]\!]) \\
\psi[\![f\ E]\!]bve & = & fve[\![f]\!](\psi[\![E]\!]bve) \\
\psi[\![(BE,E)]\!]bve & = & (\psi[\![BE]\!]bve, \psi[\![E]\!]bve) \\
\psi[\![(E,BE)]\!]bve & = & (\psi[\![E]\!]bve, \psi[\![BE]\!]bve) \\
\psi[\![BE : E]\!]bve & = & (\psi[\![BE]\!]bve) : (\psi[\![E]\!]bve) \\
\psi[\![\texttt{if}\ Cond\ \texttt{then}\ E_1\ \texttt{else}\ E_2]\!]bve & = & if\ (bottom?\ (\psi[\![Cond]\!]bve))\ then\ \perp \\
 & & elsif\ \psi[\![Cond]\!]bve\ then\ \psi[\![E_1]\!]bve\ else\ \psi[\![E_2]\!]bve \\
\psi[\![\texttt{let}\ x = E_1\ \texttt{in}\ E_2]\!]bve & = & \psi[\![E_2]\!](bve[\psi[\![E_1]\!]bve/x]) \\
\psi[\![\texttt{let}\ (x,y) = E_1\ \texttt{in}\ E_2]\!]bve & = & \psi[\![E_2]\!](bve[\psi[\![E_1]\!]bve/(x,y)]) \\
\psi[\![\texttt{let}\ x : xs = E_1\ \texttt{in}\ E_2]\!]bve & = & \psi[\![E_2]\!](bve[\psi[\![E_1]\!]bve/x : xs]) \\
bottom?\ y_1 \cdots y_n & = & (y_1 = \perp) \vee \cdots \vee (y_n = \perp)
\end{array}
$$

$$
\text{where} \left\{
\begin{array}{ll}
\psi : (Fve \rightarrow) Exp \rightarrow Bve \rightarrow D & \xi_f : Prim \rightarrow D \rightarrow D \\
\psi_p : Prog \rightarrow Fve & \xi_p : Pred \rightarrow D \rightarrow Bool \\
fve \in Fve = Fv \rightarrow D \rightarrow D & \xi_c : Const \rightarrow D \\
bve \in Bve = Bv \rightarrow D &
\end{array}
\right.
$$

Figure 4.2: Semantics of Programming Language.

An important restriction is that the conditional part of an if-expression must consist only of basic predicates without any functional symbols. Section 4.3 discusses how this requirement can be loosened. Until then, we use only null as the unary basic predicate on lists, leq, geq, and equal as the binary basic predicates on integers, : as a binary primitive function, and *nil* as a constant. The type description in a program is often omitted if it can be easily deduced by type inference.

For technical simplicity, we also set the following restrictions.

- Basic types are *Int* and *Bool*.
- Product types and recursive types are pairs and lists, respectively.
- Each function is unary, i.e., pairs must be used to compose variables.

The third restriction means that binary functions and predicates are respectively regarded as unary functions and predicates which accept the argument of pair type. This

```
fun sort as = if null as then nil else
                 let (b,bs) = max as in b:sort bs

fun max cs = let d:ds = cs in
               if null ds then (d,nil) else
                 let (e,es) = max ds in
                  if leq(e,d) then (d,e:es) else (e,d:es) end
```

Figure 4.3: Example sorting program

assumption can easily be extended to a more general setting, for instance, with tuple types.

Values are denoted by $a, b, c, \cdots$, lists by $as, bs, cs, \cdots$, and lists of lists by $ass, bss, css, \cdots$.[1] We also assume that variable names of input variables and locally defined variables are different. The functions in Fig. 4.3 present a sorting program with types

$$\begin{cases} sort : & int\ list \to int\ list \\ max : & int\ list \to int \times int\ list \end{cases}$$

4.1.2 Specification language

The language for specifying properties is constructed by using predicate constructors $\forall, \forall_l, \forall_r$, and ∇ on basic predicates, constants, free variables, and variables appearing in a program. Predicate constructors will be extended in Section 4.3. A basic predicate is denoted by p and a predicate by P. When the distinction of *unary* and *binary* is stressed, we add the index $_U$ and $_B$, respectively. As convention, bound variables are denoted by $a, b, c, \cdots, x, y, z, \cdots$, $as, bs, cs, \cdots$, $xs, ys, zs, \cdots$, free variables by $X, Y, Z, \cdots$, constants by $C, M, \cdots$, basic expressions by $BE, BE_1, BE_2, \cdots$, and expressions by $E, E_1, E_2, \cdots$.

A binary predicate P is transformed into a unary predicate P^{BE} by substituting a basic expression BE for the second argument. That is, $P^{BE}(E) = P(E, BE)$. $\bar{P}$ is defined by $\bar{P}(E_1, E_2) = P(E_2, E_1)$. The grammar of the construction of predicates is shown in Fig. 4.4, in which P_α represents a predicate on values of type α. (We will often omit types when they are clear from the context.) Specification of a function f is expressed with a free variable by

$$P(X) \to Q(f\ X)$$

which means if an input X satisfies P then an output $f\ X$ satisfies Q, when $f : \alpha \to \beta$, $P \in P^-_\alpha$, and $Q \in P^-_\beta$. Each input X is a distinct free variable for each function and property to avoid name crash.

Note that negation is not allowed except for basic predicates. The meanings and examples of the predicate constructors $\forall, \forall_l, \forall_r$, and ∇ are given as follows.

[1] *as* is a reserved word of Haskell, but we ignore it.

$P_{\alpha\to\beta}$	$=$	$P^-_\alpha\ (X) \to P^-_\beta\ (f\ X)$	properties of functions
P^-_α	$=$	predicates in P_α without bound variables	
P_α	$=$	$P_\alpha \wedge P_\alpha \mid P_\alpha \vee P_\alpha \mid P^{V_\beta}_{\alpha\times\beta} \mid \textbf{true} \mid \textbf{false}$	properties
$P_{list(\alpha)}$	$=$	$null \mid \neg null \mid \forall P_\alpha \mid \nabla P_{\alpha\times\alpha}$	properties of lists
$P_{\alpha\times\beta}$	$=$	$\bar{P}_{\beta\times\alpha} \mid P_\alpha \times P_\beta$	properties of pairs
$P_{list(\alpha)\times\beta}$	$=$	$\forall_l P_{\alpha\times\beta}$	properties of pairs
$P_{\alpha\times list(\beta)}$	$=$	$\forall_r P_{\alpha\times\beta}$	properties of pairs
$P_{Int\times Int}$	$=$	$leq \mid geq \mid equal \mid \neg P_{Int\times Int}$	basic binary predicates
V_α	$=$	$X_\alpha \mid BE_\alpha$	
BE_α			basic expressions of type α
X_α			free variables of type α

Figure 4.4: Language for specification of properties.

- $\forall P_U(xs)$ if and only if either xs is nil or the unary predicate $P_U(x)$ holds for each element x in xs.
- $\nabla P_B(xs)$ if and only if either xs is nil or $\forall \bar{P}^y_B\ (ys) \wedge \nabla P_B(ys)$ for $xs = y : ys$.

$\forall_l P_B(xs, y)$ and $\forall_r P_B(x, ys)$ are defined by $\forall P^y_B(xs)$ and $\forall \bar{P}^x_B(ys)$, respectively. The examples are shown in the table below.

predicate	*true*	*false*
$geq^3(a)$	4	2
$\forall geq^3(as)$	[3,6,4], nil	[3,6,1]
$\forall_l leq(as, a)$	([3,6,4], 8), (nil, 8)	([3,6,4], 5)
$\forall_r leq(a, as)$	(3, [3,6,4]), (3, nil)	(5, [3,6,4])
$\nabla geq(as)$	[6,4,3], nil	[4,6,3]

For instance, the sorting program is fully specified by the following conditions.

1. Output must be ordered (called **orderedness**).
2. Input and output are the same *multiset* (called **preservation**).

Orderedness is expressed as $\textbf{true} \to \nabla geq(sort\ X)$. That is, the output of the sorting program is decreasing if the input satisfies **true** (i.e., empty assumptions). For preservation, the weaker condition called weak preservation, i.e., the input and the output are the same *set*, is expressed by

$$\textbf{true} \to (\forall_l \exists_r equal \wedge \forall_r \exists_l equal)^X (sort\ X).$$

with the introduction of additional predicate constructors $\exists$, $\exists_l$, and $\exists_r$ (which intuitively mean $\neg\forall\neg$, $\neg\forall_l\neg$, and $\neg\forall_r\neg$, respectively), which will be discussed in Section 4.3. Note that only the input variable of a function remains in the scope when a function call occurs, thus our definition of properties of functions (P_F) is possible (which was neglected in [LM95]).

4.2 Automatic verification as abstract interpretation

4.2.1 Verification algorithm as abstract interpretation

An abstract interpretation consists of an abstract domain, its order, and an interpretation (on an abstract domain) of primitive functions [CC77, AH87, Bur91]. Our choice is a backward abstract interpretation with

abstract domain	$Pred = \cup P_\alpha$
order	$\sqsubseteq$ (the entailment relation)
interpretation	$\Psi : Exp \to Pred \to Bvp$ where $Bvp = P[Bv \to Pred]$.

P_α, $\sqsubseteq$, and Ψ are formerly defined in Fig. 4.4, Fig. 4.5, and Fig. 4.6, respectively. Then, for a program { `fun` $f_i\ x_i = E_i$; }, the verification algorithm is the least fixed point computation to solve *whererec* equations in a backward manner.

In Fig. 4.6, an element in Bvp is regarded as a disjunctive canonical form in which each predicate (in Fig. 4.4) accepts only bound variables (in the scope of an expression) and constants as an instance. The disjunction $\vee$ is introduced when analyzing conditional expressions, and is regarded as composing branches of the verification, i.e., each branch (conjunctive formula) is analyzed independently. The conjunction $\wedge$ is regarded as an assignment from bound variables to predicates. Section 4.3 explains how this algorithm performs on the sorting program.

The entailment relation $\sqsubseteq$ (in Fig. 4.5) is intuitively the opposite of the logical implication. That is, $P \sqsubseteq Q$ means that Q *implies* P. By definition, **true** is the least element and **false** is the greatest element in the abstract domain. We denote by $P \equiv Q$ if $P \sqsubseteq Q$ and $P \sqsupseteq Q$. The entailment relation may be used to trim at each step of interpretation Ψ. Formally, the entailment relation consists of axioms on basic predicates/predicate constructors, and ordinary logical rules, and their extensions by predicate constructors, as defined in Fig. 4.5.

The projection $\gamma \downarrow_x$ extracts a unary predicate P_U whose instance is a variable $x \in Bv$ from a conjunctive formula γ, and the projection $\gamma \downarrow_{(x,y)}$ extracts a binary predicate P_B whose instance is a pair of variables $(x, y) \in Bv \times Bv$. For instance, $(leq(x,y) \wedge \nabla(xs)) \downarrow_x = leq^y(x)$. For a conjunctive formula $\gamma = P_1(x_1) \wedge \cdots \wedge P_k(x_k)$, a predicate P, and a bound variable x, we define

$$\gamma \setminus P(x) = \begin{cases} P_1(x_1) \wedge \cdots \wedge P_{i-1}(x_{i-1}) \wedge P_{i+1}(x_{i+1}) \wedge \cdots \wedge P_k(x_k) & \text{if } P_i(x_i) \equiv P(x) \\ P_1(x_1) \wedge \cdots \wedge P_k(x_k) & \text{otherwise} \end{cases}$$

The convex-hull operator $\ominus$ (which eliminates $\neg$) is defined by

$$\ominus[\ (Cond \wedge P) \vee (\neg Cond \wedge P')\ ] = \vee_i Q_i$$

where each Q_i satisfies $Cond \wedge P \sqsubseteq Q_i$ and $\neg Cond \wedge P' \sqsubseteq Q_i$. For instance, $\ominus[\ (leq(e,d) \wedge \forall leq^d(es)) \vee (\neg leq(e,d) \wedge \forall leq^e)\ ] = \forall leq^d(es) \vee \forall leq^e(es)$.

A special treatment is required on the interpretation of a function call

$$\Psi[\![f\ E]\!]P = \Psi[\![E]\!]((fvp[\![f]\!]P^-)\theta^-)$$

Axioms on basic predicates

$$\overline{equal} \equiv equal \qquad \overline{leq} \equiv geq \qquad null(\texttt{nil}) \equiv \textbf{true} \qquad null(x:xs) \equiv \textbf{false}$$

$$equal(x,x) \equiv \textbf{true} \qquad geq(x,x) \equiv \textbf{true} \qquad leq(x,x) \equiv \textbf{true}$$

$$equal \sqsubseteq equal^x \times \overline{equal}^x \qquad geq \sqsubseteq geq^x \times \overline{geq}^x \qquad leq \sqsubseteq leq^x \times \overline{leq}^x$$

$$\frac{P \sqsubseteq P^x \times \bar{P}^x}{\forall P^y \sqsubseteq \forall P^z \wedge P(z,y)} \qquad \frac{P \sqsubseteq P^x \times \bar{P}^x}{\forall_l\forall_r P \sqsubseteq (\forall_l\forall_r P)^{xs} \times (\overline{\forall_l\forall_r P})^{xs}}\ \textbf{Transitivity}$$

Ordinary logical rules on logical connectives

$$P \wedge P \equiv P \qquad P_1 \sqsubseteq P_1 \wedge P_2 \qquad \textbf{true} \wedge P \equiv P \qquad \textbf{false} \wedge P \equiv \textbf{false}$$

$$P \vee P \equiv P \qquad P_1 \vee P_2 \sqsubseteq P_1 \qquad \textbf{true} \vee P \equiv \textbf{true} \qquad \textbf{false} \vee P \equiv P$$

$$\frac{P_1 \sqsubseteq P_2 \qquad P_1' \sqsubseteq P_2'}{P_1 \wedge P_1' \sqsubseteq P_2 \wedge P_2'} \qquad \frac{P_1 \sqsubseteq P_2 \qquad P_1' \sqsubseteq P_2'}{P_1 \vee P_1' \sqsubseteq P_2 \vee P_2'}$$

Entailment relation of predicate constructors

$$\frac{P_1 \sqsubseteq P_2}{\bar{P}_1 \sqsubseteq \bar{P}_2} \qquad \frac{P_1 \sqsubseteq P_2}{\dagger P_1 \sqsubseteq \dagger P_2}\ \textbf{list} \qquad \dagger\,(P_1 \wedge P_2) \equiv \dagger P_1 \wedge \dagger P_2 \quad \text{with } \dagger \in \{\forall, \forall_l, \forall_r, \nabla\}$$

$$\frac{P_1 \sqsubseteq P_1' \qquad P_2 \sqsubseteq P_2'}{P_1 \times P_2 \sqsubseteq P_1' \times P_2'}\ \textbf{pair} \qquad (P_1 \times P_2) \wedge (P_1' \times P_2') \equiv (P_1 \wedge P_1') \times (P_2 \wedge P_2')$$

$$\overline{\overline{P}} \equiv P \qquad \overline{P_1 \wedge P_2} \equiv \bar{P}_1 \wedge \bar{P}_2 \qquad \overline{P_1 \vee P_2} \equiv \bar{P}_1 \vee \bar{P}_2 \qquad \overline{P_1 \times P_2} \equiv P_2 \times P_1$$

$$\overline{\forall_l P} \equiv \forall_r \bar{P} \qquad \overline{\forall_r P} \equiv \forall_l \bar{P} \qquad (\forall_l P)^V \equiv \forall(P^V) \qquad \forall_l\forall_r P \equiv \forall_r\forall_l P$$

$$\forall P(a:as) \equiv P(a) \wedge \forall P(as) \qquad \forall P(nil) \equiv \textbf{true} \qquad P^y(x) \equiv \bar{P}^x(y)$$

$$\nabla P(a:as) \equiv \forall \bar{P}^a(as) \wedge \nabla P(as) \qquad \nabla P(nil) \equiv \textbf{true}$$

Figure 4.5: Entailment relation

$$
\begin{aligned}
\Phi\llbracket\ \{\ \texttt{fun}\ f_i\ x_i = E_i\ ;\ \}\ \rrbracket &= fvp \text{ whererec } fvp = [\ \lambda P_1 \cdots P_n . \Psi\llbracket E_i \rrbracket P_i / f_i\ ] \\
\Psi\llbracket C \rrbracket P &= \begin{cases} \textbf{true} & \text{if } P(C) \\ \textbf{false} & \text{otherwise} \end{cases} \\
\Psi\llbracket x \rrbracket P &= P(x) \\
\Psi\llbracket op\ E \rrbracket P &= \Psi\llbracket E \rrbracket (\Xi\llbracket op \rrbracket P) \\
\Psi\llbracket f\ E \rrbracket P &= \Psi\llbracket E \rrbracket ((fvp\llbracket f \rrbracket P^{-})\theta^{-}) \\
\Psi\llbracket (BE, E) \rrbracket P &= \begin{cases} \Psi\llbracket BE \rrbracket P_1 \wedge \Psi\llbracket E \rrbracket P_2 & \text{if } P = P_1 \times P_2 \\ \Psi\llbracket E \rrbracket \bar{P}^{BE} & \text{otherwise} \end{cases} \\
\Psi\llbracket (E, BE) \rrbracket P &= \begin{cases} \Psi\llbracket E \rrbracket P_1 \wedge \Psi\llbracket BE \rrbracket P_2 & \text{if } P = P_1 \times P_2 \\ \Psi\llbracket E \rrbracket P^{BE} & \text{otherwise} \end{cases} \\
\Psi\llbracket BE : E \rrbracket P &= \begin{cases} \Psi\llbracket BE \rrbracket Q \wedge \Psi\llbracket E \rrbracket \forall Q & \text{if } P = \forall Q \\ \Psi\llbracket (BE, E) \rrbracket \forall_r Q \wedge \Psi\llbracket E \rrbracket \nabla Q & \text{if } P = \nabla Q \end{cases} \\
\Psi\llbracket \texttt{if}\ Cond\ \texttt{then}\ E_1\ \texttt{else}\ E_2 \rrbracket P &= \ominus[(Cond \wedge \Psi\llbracket E_1 \rrbracket P) \vee (\neg Cond \wedge \Psi\llbracket E_2 \rrbracket P)] \\
\Psi\llbracket \texttt{let}\ x = E_1\ \texttt{in}\ E_2 \rrbracket P &= \Psi\llbracket E_1 \rrbracket (\Psi\llbracket E_2 \rrbracket P){\downarrow_x}\ \wedge (\Psi\llbracket E_2 \rrbracket P \setminus (\Psi\llbracket E_2 \rrbracket P){\downarrow_x}) \\
\Psi\llbracket \texttt{let}\ (x,y) = E_1\ \texttt{in}\ E_2 \rrbracket P &= \Psi\llbracket E_1 \rrbracket (\Psi\llbracket E_2 \rrbracket P){\downarrow_{(x,y)}}\ \wedge (\Psi\llbracket E_2 \rrbracket P \setminus (\Psi\llbracket E_2 \rrbracket P){\downarrow_{(x,y)}}) \\
\Psi\llbracket \texttt{let}\ x : xs = E_1\ \texttt{in}\ E_2 \rrbracket P &= \begin{cases} \Psi\llbracket E_1 \rrbracket \nabla Q\ \wedge\ \Psi\llbracket E_1 \rrbracket (\Psi\llbracket E_2 \rrbracket P \setminus \forall_r Q(x, xs)) \\ \qquad\qquad \text{if } (\Psi\llbracket E_2 \rrbracket P){\downarrow_{(x,xs)}} \sqsupseteq \forall_r Q \\ \Psi\llbracket E_1 \rrbracket \forall Q\ \wedge\ \Psi\llbracket E_1 \rrbracket (\Psi\llbracket E_2 \rrbracket P \setminus Q(x)) \\ \qquad\qquad \text{if } (\Psi\llbracket E_2 \rrbracket P){\downarrow_x} \sqsupseteq Q \\ \Psi\llbracket E_1 \rrbracket \forall Q\ \wedge\ \Psi\llbracket E_1 \rrbracket (\Psi\llbracket E_2 \rrbracket P \setminus \forall Q(xs)) \\ \qquad\qquad \text{if } (\Psi\llbracket E_2 \rrbracket P){\downarrow_{xs}} \sqsupseteq \forall Q \\ \Psi\llbracket E_2 \rrbracket P \qquad\qquad \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\text{where} \begin{cases} \Psi : (Fvp \rightarrow) Exp \rightarrow Pred \rightarrow Bvp & \Xi : Prim \rightarrow Pred \rightarrow Pred \\ \Phi : Prog \rightarrow Fvp & \downarrow : Bvp \rightarrow Bv \rightarrow Pred \\ fvp \in Fvp = Exp \rightarrow Pred \rightarrow Pred & \ominus : Bvp \rightarrow Bvp \\ bvp \in Bvp = P[Bv \rightarrow Pred] & \end{cases}
$$

Figure 4.6: Abstract semantics of verification

$$\begin{tikzcd}
D_\alpha \arrow[r, "f_{\alpha\to\beta}"] \arrow[d, "abs^1_\alpha"', "(cl_{D_\alpha})"] & D_\beta \\
PD[D_\alpha] & PD[D_\beta] \arrow[l, "abs^1_\alpha \circ f^{-1} \circ conc^1_\beta"] \arrow[u, "conc^1_\beta", "(Max_{D_\beta})"']
\end{tikzcd}
\qquad
\begin{tikzcd}
PD[D_\alpha] \arrow[d, "abs^2_\alpha"'] & PD[D_\beta] \arrow[l, "abs^1_\alpha \circ f^{-1} \circ conc^1_\beta"'] \\
Pred_\alpha & Pred_\beta \arrow[l, "abs^2_\alpha \circ abs^1_\alpha \circ f^{-1} \circ conc^1_\beta \circ conc^2_\beta"] \arrow[u, "conc^2_\beta"']
\end{tikzcd}$$

Figure 4.7: Two steps of domain-induced abstract interpretation

That is, when a function call of f occurs, a predicate P may contains some bound variables. However, except for the input of f, all bound variables except for an input variable become out of scope. Thus, a predicate P is separated to a predicate P^-, which is obtained by replacing each bound variable x in P with a free variable Y, and a substitution θ^-, which is a collection of $[Y \leftarrow x]$, such that $P = P^-\theta^-$. For instance, when $\Psi[\![f\ E]\!]\forall leq^b$ is analyzed, $\forall leq^b$ is separated to a predicate $P^- = \forall leq^Z$ and a substitution $\theta^- = [Z \leftarrow b]$.

Theorem 4.1 *The verification algorithm always terminates.*

(Sketch of proof) Basic predicates, variables, and constants appearing in a program are finite. A free variable is introduced as a substitute for an expression only when function-calls; thus, only finitely many free variables are used during verifications. Since each predicate constructor enriches types, once types of functions are fixed only finitely many applications of predicate constructors are possible. The finiteness of an input-dependent abstract domain is then guaranteed. The algorithm is therefore formulated as the least fix-point computation on a finite abstract domain, so that it terminates. ■

4.2.2 Soundness by domain-induced abstract interpretation

In this section, we will show how the abstract interpretation Φ is obtained as a domain-induced abstract interpretation, i.e., an abstract interpretation induced from domain abstractions. As a consequence, the soundness proof is given. Note that an automatic verification cannot be complete by nature.

Let the domain and codomain of a function f of type $\alpha \to \beta$ be D_α and D_β, respectively. Let the power domain $PD[D_\alpha]$ of D_α be $\{cl_{D_\alpha}(\mathcal{X}) \mid \mathcal{X} \subseteq D_\alpha\}$ with the order $\sqsubseteq_{-1} = \supseteq$, where cl_{D_α} is the downward closure operator in D_α.

Φ (in Fig. 4.6) is expressed as the two-step domain-induced abstract interpretation (as indicated in Fig. 4.7). The first step is backward and consists of

- the abstract domain $PD[D_\alpha]$
- the concretization map $conc^1_\alpha \ = \ Max_{D_\alpha}$

- the abstraction map $abs^1_\alpha \;=\; cl_{D_\alpha}$

This step precisely detects how much of the input is enough to produce the output satisfying the specification. The next step approximates according to the specification language in order to make the analysis decidable. Let $Pred_\alpha$ be a set of predicates on D_α generated as P^-_G in Fig. 4.4. The second step is forward and consists of

- the abstract domain $Pred_\alpha$.
- the concretization map $conc^2_\alpha(P) \;=\; cl_{D_\alpha}(\{x \in D_\alpha \mid P(x)\})$ for $P \in Pred_\alpha$.
- the abstraction map $abs^2_\alpha(\mathcal{X}) \;=\; \sqcap(\{P \in Pred_\alpha \mid conc^2_\alpha(P) \subseteq \mathcal{X}\})$ for $\mathcal{X} \in PD[D_\alpha]$.

Note that the abstract domain $Pred_\alpha$ is a lattice wrt the entailment relation. For instance, $P \sqcup Q$ and $P \sqcap Q$ always exists as $P \wedge Q$ and $P \vee Q$, respectively.

Thus an abstract interpretation Ξ on a primitive function *op* of type $\alpha \to \beta$ is defined by $\Xi(op) = abs_\alpha \cdot op^{-1} \cdot conc_\beta$, where $abs_\alpha = abs^2_\alpha \cdot abs^1_\alpha$ and $conc_\beta = conc^1_\beta \cdot conc^2_\beta$. Similar to Ψ on expressions. The abstract interpretation Φ on recursively defined functions f_i's is obtained by the least fix-point computation.

Definition 4.2 *For an abstract interpretation Φ, a function f is safe if f satisfies $abs \cdot f^{-1} \cdot conc \sqsubseteq \Phi(f)$. An abstract interpretation Ψ is safe if each primitive function is safe.*

Theorem 4.3 *The verification algorithm is sound (i.e., the detected property of a program always holds if a program terminates).*

(Sketch of proof) From definition in Fig. 4.6, Ψ is safe. Let f be a function defined by a recursive equation R, and let f' be a function defined by a recursive equation $\Psi(R)$. Then, $f = Fix(R) = \sqcup\{f_i\}$ and $\Phi(f) = f' = Fix(\Psi(R)) = \sqcup\{f'_i\}$ where $f_i = R^i(\perp)$ and $f'_i = (\Psi(R))^i(\perp)$.

Since Ψ is safe and $(abs, conc)$ is the dual of Galois connection (i.e., $abs_\alpha \cdot conc_\alpha = id_{D_\alpha}$ and $conc_\alpha \cdot abs_\alpha \subseteq id_{D_\alpha}$), thus $abs \cdot f_i^{-1} \cdot conc \sqsubseteq f'_i$.

It is not difficult to show $abs \cdot \sqcup f_i^{-1} \cdot conc \sqsubseteq \sqcup(abs \cdot f_i^{-1}) \cdot conc \sqsubseteq \sqcup(abs \cdot f_i^{-1} \cdot conc)$, thus $abs \cdot f^{-1} \cdot conc = abs \cdot \sqcup f_i^{-1} \cdot conc \sqsubseteq \sqcup(abs \cdot f_i^{-1} \cdot conc) \sqsubseteq \sqcup f'_i = f' = \Psi(f)$. ■

4.2.3 Related work

Many studies have been undertaken on verification. Most are based on theorem provers, for example, Coq, LCF, Boyer-Moore prover, Larch, and EQP. They require either complex heuristics or strong human guidance (or both), either of which is not easy to learn. However, for huge, complex, and critical systems, this price is worth paying.

The complementary approach uses intelligent compile-time error detection for easy debugging. For imperative programs, Bourdoncle proposed an assertion-based debugging called *Abstract debugging* [Bou93b, Bou93a]. For logic programs, Comini, et. al. [CLMV96] and Bueno, et. al. [B$^+$97] proposed extensions of declarative diagnosis based on abstract interpretation. Cortesi, et. al. [CLCR96, LCLRC97] proposed the automatic verification based on abstract interpretation. Levi and Volpe proposed the framework based on abstract interpretation to classify various verification methods [LV98]. Among them, target specifications primarily focus on behavior properties, such as termination, mutual exclusion of clauses, and size/cardinality relation between inputs and outputs.

In contrast, Métayer's [LM95] and our specification language (for functional programs) directly express the programmer's intention in a concise and declarative description. This point is more desirable for some situation, such as, when a novice programmer writes a relatively small program.

As an abstract interpretation, our framework is similar to *inverse image analysis* [Dyb91]. The significant difference is that inverse image analysis determines *how much of the input is needed to produce a certain amount of output* and computes Scott's *open* sets. Our framework, in contrast, determines *how much of the input is enough to produce a certain amount of output* and computes Scott's *closed* sets. In terms of [OO91b], the former is expressed by a HOMT $((id, id), -, (\sqsubseteq_{-0}, RC), Min)$, and the latter is expressed by $((id, id), -, (\sqsubseteq_{-1}, LC), Max)$.

Similar techniques that treat abstract domain construction as a set of predicates are found in several places. In [EM91, Jen94, Ben93, OO91a], predicates are either limited to unary (such as *null* and $\neg null$), or in [CFW91] predicates are limited to propositions corresponding to variables appearing in a (logic) program. Refs. [HPS96, XP98, XP99, CK00] adopt Presberger arithmetic (in which the satisfiability is decidable) in type-based analyses. That is, data structures are projected to their sizes, whereas our approach works on the properties reflecting structures of recursive types. The combination with our method would be the possible direction of extensions.

4.3 Examples and extensions

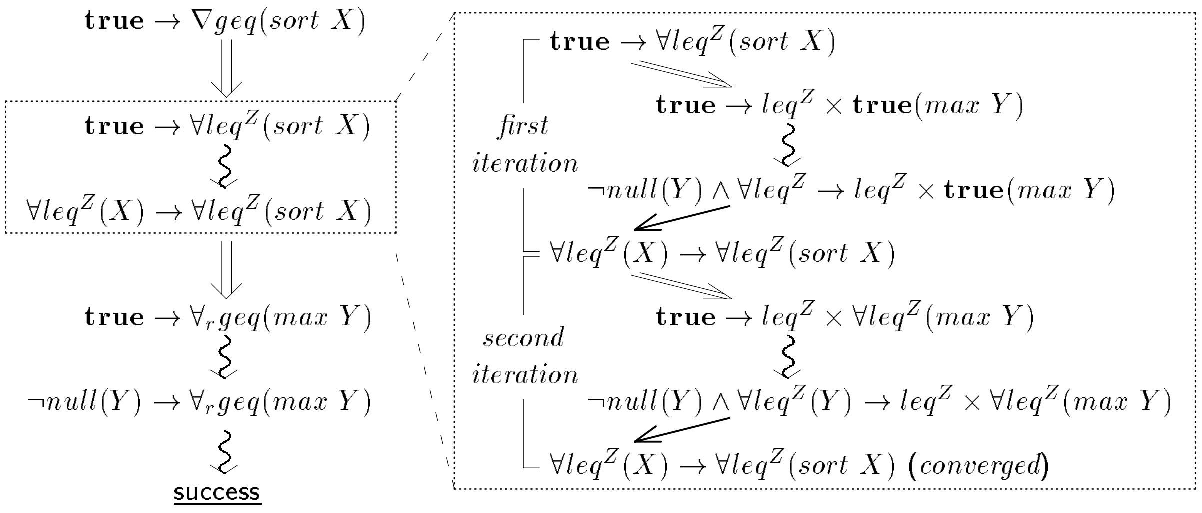
4.3.1 Verifying orderedness of sorting

The verification algorithm is explained here by an example of orderedness $\mathbf{true} \to \nabla geq(sort\ X)$. When unknown properties of user-defined functions are required, new conjectures are produced. For instance, when verifying $\mathbf{true} \to \nabla geq(sort\ X)$, it automatically produces and proves the lemmata; $\forall leq^Z(X) \to \forall leq^Z(sort\ X)$, $\neg null \wedge \forall leq^Z(Y) \to leq^Z \times \forall leq^Z(max\ Y)$, and $\neg null(Y) \to \forall_r geq(max\ Y)$. The generation of lemmata is shown at the top of Fig. 4.8. The vertical wavy arrow indicates an iterative procedure, the double arrow indicates the creation of a conjecture, and the arrow returns the resulting lemma.

For instance, $\forall leq^Z(X) \to \forall leq^Z(sort\ X)$ means that if an input of *sort* is *less-than-or-equal-to* any given Z, an output is also *less-than-or-equal-to* Z. This lemma is generated as a conjecture $\mathbf{true} \to \forall leq^Z(sort\ X)$ at the `else`-branch of the `if`-expression in `sort` ($\Psi[\![$`b:sort bs`$]\!]\nabla geq$) as follows.

$$\begin{array}{rcl} \nabla geq(b : sort\ bs) & \equiv & \forall_r geq(b, sort\ bs) \ \wedge\ \nabla geq(sort\ bs) \\ & \equiv & \forall leq^b(sort\ bs) \ \wedge\ \nabla geq(sort\ bs) \end{array}$$

Since there are no conjectures related to $\forall leq^b(sort\ X)$ in the recursion hypothesis, a new conjecture is created. But properties of functions (P_F in Fig. 4.4) exclude bound



$\mathbf{true} \to \nabla geq(sort\ X)$
$\mathbf{true} \to \forall leq^Z(sort\ X)$
$\forall leq^Z(X) \to \forall leq^Z(sort\ X)$
$\mathbf{true} \to \forall_r geq(max\ Y)$
$\neg null(Y) \to \forall_r geq(max\ Y)$
success
first iteration
$\mathbf{true} \to \forall leq^Z(sort\ X)$
$\mathbf{true} \to leq^Z \times \mathbf{true}(max\ Y)$
$\neg null(Y) \wedge \forall leq^Z \to leq^Z \times \mathbf{true}(max\ Y)$
$\forall leq^Z(X) \to \forall leq^Z(sort\ X)$
second iteration
$\mathbf{true} \to leq^Z \times \forall leq^Z(max\ Y)$
$\neg null(Y) \wedge \forall leq^Z(Y) \to leq^Z \times \forall leq^Z(max\ Y)$
$\forall leq^Z(X) \to \forall leq^Z(sort\ X)$ (*converged*)

Figure 4.8: Generation of lemmata for $\mathbf{true} \to \nabla geq(sort\ X)$

variables. Thus, by separating a predicate $\forall leq^b$ to $\forall leq^Z$ and the substitution $[Z \leftarrow b]$, $\mathbf{true} \to \forall leq^Z(sort\ X)$ is created. This means that no local information on b is used during the verification of $\mathbf{true} \to \forall leq^Z(sort\ X)$. This conjecture does not hold; instead, we obtain $\forall leq^Z(X) \to \forall leq^Z(sort\ X)$ as a lemma.

A typical example of the use of the entailment relation appears in the verification of $\neg null(Y) \to \forall_r geq(max\ Y)$. At the second `if`-expression in `max`,

$$\Psi[\![\texttt{if leq(e,d) then (d,e:es) else (e,d:es)}]\!]\ \forall_r geq$$

is created. Thus, $(leq(e,d) \wedge \forall leq^d(es))\ \vee\ (\neg leq(e,d) \wedge \forall leq^e(es))$ is obtained. From the transitivity of leq, $leq(e,d) \wedge \underline{\forall leq^d(es)} \sqsubseteq leq(e,d) \wedge \underline{\forall leq^e(es)}$ (see the underlined parts), therefore we obtain $\forall leq^e(es)$ by the convex-hull operation. Note that the convex-hull operation also creates $\forall leq^d(es)$, but only $\forall leq^e(es)$ branch is successful, i.e., from the recursion hypothesis $\Psi[\![\texttt{max ds}]\!]\forall_r geq$ is reduced to $\neg null(ds)$, as desired. Thus $\forall leq^d(es)$ is omitted.

4.3.2 New predicate constructors: verifying weak preservation of sorting

In this section we introduce new predicate constructors and extend the entailment relation to make it possible to verify weak preservation of sort programs. The new predicate constructors are $\exists, \exists_l, \exists_r$, and $\triangle$. The predicates are extended by updating part of the grammar in Fig. 4.4 with

$$\begin{array}{lcll}
P_{list(\alpha)} & = & null \mid \forall P_\alpha \mid \underline{\exists P_\alpha} \mid \nabla P_{\alpha\times\alpha} \mid \underline{\triangle P_{\alpha\times\alpha}} & \text{properties of lists} \\
P_{list(\alpha)\times\beta} & = & \forall_l P_{\alpha\times\beta} \mid \underline{\exists_l P_{\alpha\times\beta}} & \text{properties of pairs} \\
P_{\alpha\times list(\beta)} & = & \forall_r P_{\alpha\times\beta} \mid \underline{\exists_r P_{\alpha\times\beta}} & \text{properties of pairs}
\end{array}$$

$$\frac{P_1 \sqsubseteq P_2}{\triangle P_1 \sqsubseteq \triangle P_2} \qquad \frac{P_1 \sqsubseteq P_2}{\dagger P_1 \sqsubseteq \dagger P_2}\ \textbf{list} \qquad \dagger(P_1 \vee P_2) \equiv \dagger P_1 \vee \dagger P_2 \quad \text{with } \dagger \in \{\exists, \exists_l, \exists_r\}$$

$$\overline{\exists_l P} \equiv \exists_r \bar{P} \qquad \overline{\exists_r P} \equiv \exists_l \bar{P} \qquad (\exists_l P)^E \equiv \exists(P^E) \qquad \exists_l \exists_r P \equiv \exists_r \exists_l P$$

$$\frac{P \sqsubseteq P^x \times \bar{P}^x}{\forall_l \exists_r P \sqsubseteq (\forall_l \exists_r P)^{xs} \times (\overline{\forall_l \exists_r P})^{xs}} \qquad \frac{P \sqsubseteq P^x \times \bar{P}^x}{\exists_l \forall_r P \sqsubseteq (\exists_l \forall_r P)^{xs} \times (\overline{\exists_l \forall_r P})^{xs}}\ \textbf{Transitivity}$$

$$\begin{array}{rclrcl} \exists P(a:as) & \equiv & P(a) \vee \exists P(as) & \forall_l \exists_r P(as, b:bs) & \sqsubseteq & \forall_l \exists_r P(as, bs) \\ \exists P(nil) & \equiv & \textbf{false} & \forall_r \exists_l P(a:as, bs) & \sqsubseteq & \forall_r \exists_l P(as, bs) \end{array}$$

$$\triangle P(a:b:bs) \equiv \exists \bar{P}^a(b:bs) \wedge (null(bs) \vee \triangle P(b:bs))$$

$$\triangle P(a:nil) \equiv \textbf{false} \qquad \triangle P(nil) \equiv \textbf{false}$$

Figure 4.9: New entailment relation

where the underlined parts are newly added. Their meanings are shown by examples in the table below. The entailment relation is enriched as in Fig. 4.9.

predicate	*true*	*false*
$\exists geq^5(as)$	[3,6,4]	([3,2,4]), nil
$\exists_l leq(as, a)$	([3,6,4], 5)	([3,6,4], 2), (nil, 5)
$\exists_r leq(a, as)$	(5, [3,6,4])	(7, [3,6,4]), (5, nil)
$\triangle leq(as)$	[3,2,4]	[3,6,4], [3], nil

Then weak preservation of `sort` is expressed by

$$\textbf{true} \rightarrow \overline{\forall_l \exists_r equal \wedge \forall_r \exists_l equal}^X (sort\ X).$$

During the verification of $\textbf{true} \rightarrow \overline{\forall_l \exists_r equal}^X (sort\ X)$, the key step is at $\forall_l \exists_r equal(as, b : sort(bs))$ in `sort`. By transitivity $\forall_l \exists_r equal(as, b:bs) \wedge \forall_l \exists_r equal(b:bs, b:sort(bs))$ is inferred. To solve the second component, the entailment relation $\forall_l \exists_r P(a:as, b:bs) \sqsubseteq P(a,b) \wedge \forall_l \exists_r P(as, bs)$ is used. This is obtained as a transitive closure by

$$\begin{array}{rcl} \forall_l \exists_r P(a:as, b:bs) & \equiv & \exists_r P(a, b:bs) \wedge \underline{\forall_l \exists_r P(as, b:bs)} \\ & \sqsubseteq & (P(a,b)\ \underline{\vee\ \exists_r P(a, bs)}) \wedge \forall_l \exists_r P(as, bs) \\ & \sqsubseteq & P(a,b) \wedge \forall_l \exists_r P(as, bs). \end{array}$$

Thus $\forall_l \exists_r equal(b:bs, b:sort(bs))$ is reduced to $\forall_l \exists_r equal(bs, sort(bs))$ which is a recursion hypothesis. The rest $\forall_l \exists_r equal(as, b:bs)$ creates the conjecture

$$\textbf{true} \rightarrow \overline{\forall_l(equal \times \textbf{true} \vee \textbf{true} \times \exists_r equal)}^Y (max\ Y)$$

at `(b,bs) = max as`, and similar approximations occur in `max` at expressions `(d,e:es)` and `(e,d:es)`. $\textbf{true} \rightarrow \overline{\exists_l \forall_r equal}^X (sort\ X)$ is similarly verified.

```
fun format as = f (as,nil);

fun f (bs,cs) = if null bs then [cs] else
                  let d:ds = bs
                   in if leq (width (cs++[d]),M)
                         then f (ds,cs++[d]))
                         else cs:f (ds,[d])

fun width es = if null es then 0 else
                 let f:fs=es in
                  if null fs then size f
                             else 1+(size f)+(width fs)
```

Figure 4.10: Example format program

4.3.3 Uninterpreted functions and predicates

This section extends the range of conditional expressions that can be included in the programs to be verified. Function symbols (either primitive or user-defined) in the conditional part of an `if`-expression are allowed. They are left uninterpreted during the verification, and the result will be refined by partial evaluation of these function symbols.

The example is a formatting program `format` (shown in Fig. 4.10) that formats a sentence (expressed by a list of strings) as a list of sentences each of which has a width *less-than-or-equal-to* a specified number M. Its specifications are as follows.

- Each sentence of the output must have a width less-than-equal to M.
- The order of each word in the input must be kept in the output.
- Each word of the input must appear in the output, and vice versa.

In this example, *string* is added to base types. Basic functions $+$ and constants $0, 1$ also are used in the program, but they are not directly related to the verification. Thus, their interpretation and entailment relations are omitted.

The first specification of `format` states that an output must satisfy $\forall(leq^M \cdot width)$. Note that this predicate allows a function symbol $width$ in it. Verification starts with $\mathbf{true} \rightarrow \forall(leq^M \cdot width)(format\ X)$, which is immediately reduced to $\mathbf{true} \rightarrow \forall(leq^M \cdot width)(f\ Y)$. The result of the verification is

$$(\forall leq^M \cdot width \cdot [\,]) \times (leq^M \cdot width)(Y) \rightarrow \forall(leq^M \cdot width)(f\ Y),$$

and this deduces

$$(\forall leq^M \cdot width \cdot [\,])(X) \wedge (leq^M \cdot width)(nil) \rightarrow \forall(leq^M \cdot width)(format\ X).$$

Note that the result is not affected by whatever $width$ is, since $width$ is left uninterpreted. The key steps are at `if leq (width (cs++[d]),M) then ...`. Since the throughout of `then`-branches $leq(width\ (cs++[d]), M)$ holds, the second argument of $f\ (ds, cs++[d])$ always satisfies $leq^M \cdot width$. These steps depend only on the convex-hull operation so that the function symbol *width* remains uninterpreted.

With the aid of partial evaluation which leads to $width\ nil = 0$, $width\ [x] = size\ x$, we obtain $\forall(leq^M \cdot size)(X) \wedge leq(0, M) \rightarrow \forall(leq^M \cdot width)(format\ X)$. For the partial evaluation, only the relation between *size* and *width* is important. The information on the function *size* is required only when the final result above is interpreted by a human being. Note that in general a partial evaluation may not terminate. However, this final step is devoted to transforming the detected property into a more intuitive form for a human being, and even if it fails the detected property is correct.

The second and the third specification of `format` are similar to orderedness and weak preservation of `sort`, respectively. They require further extensions.

The second specification of `format` is expressed by a *fresh* binary predicate *Rel* on pairs of strings as $\nabla Rel\ (X) \rightarrow \forall\nabla Rel \wedge \nabla\Box Rel\ (format\ X)$ where $\Box$ is an abbreviation of $\forall_l\forall_r$. Note that throughout the verification the predicate *Rel* is left uninterpreted. This implies that the specification above holds for any binary relation *Rel*. Finally, the meaning of *Rel* is assumed by a human being, and in this case it is suitable to be interpreted as the appearance order of strings.

The third specification is expressed by **true** $\rightarrow \overline{\forall_l\exists_r\exists_r equal}^X (format\ X)$ and **true** $\rightarrow \overline{\forall_r\forall_r\exists_l equal}^X (format\ X)$. Our algorithm detects the latter, but for the former we also need a new transitivity-like entailment relation of type $list(\alpha) \times list(list(\alpha))$, i.e.,

$$\frac{P \sqsubseteq P^x \times \bar{P}^x}{\forall_l\exists_r\exists_r P \sqsubseteq (\forall_l\exists_r P)^{xs} \times (\overline{\forall_l\exists_r\exists_r P})^{xs}}.$$

4.4 Termination criteria beyond finiteness

The termination of abstract interpretation has been guaranteed by the finiteness of abstract domains. For instance, we discussed the instance of CPAs over finite domain in Chapter 3. These standard abstract interpretations have the fixed abstract domains; but we can also postpone to fix an abstract domain until an instance of the input is given. For instance, the verification presented in this chapter generates a finite abstract domain corresponding to variable names in an input program. Thus, the size of the abstract domain is unbounded in general, but still finite for each instance.

There have been several trials to extend an abstract interpretation to infinite domains [Hal90, Jag89]; however, they rely on either lazy evaulation or inductive methods, and did not guarantee termination. Cousot and Cousot proposed *widening*, and mentioned that the ascending chain condition guarantees the termination [CC92].

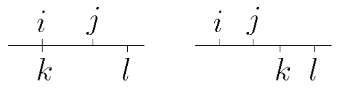
The aim of last section is to give a new termination criteria for a backward abstract interpretation in terms of *better-quasi-order* (BQO), which is a sufficient condition for the ascending chain condition: If an abstract domain is better-quasi-ordered, then its power domain (either with $\sqsubseteq_1$ or $\sqsubseteq_0$) is also better-quasi-ordered. Therefore, a backward abstract interpretation associated with either $\sqsubseteq_{-0}$ (such as a backward stractness analysis in Chapter 3) or $\sqsubseteq_{-1}$ (such as verficication in Chapter 4) will terminate.

4.4.1 Better quasi prder

BQO is the extension of WQO intended for infinite objects. For example, consider Higman's lemma, which tells that the embedding over finite words is WQO if the base domain is WQO. This is not true for infinite words as Rado's example shows [Rad54].

Rado's example Let $A = \{(i,j) \mid 0 \le i < j\}$.

$(i,j) \le (k,l)$ if and only if either $i = k \wedge j \le l$ or $j \le k$.



$\alpha_1, \alpha_2, \cdots$ is a *bad sequence* where

$$\begin{array}{rcc}
\alpha_1 & = & \langle (0,1),(1,2),(1,3),(1,4),\cdots\rangle \\
\alpha_2 & = & \langle (0,1),(1,2),(2,3),(2,4),\cdots\rangle \\
\cdots & = & \cdots \\
\alpha_i & = & \langle (0,1),\cdots,(i,i+1),(i,i+2),(i,i+3),\cdots\rangle
\end{array}$$

The difference between WQO and BQO is that the former uses a map from natural numbers (to express a sequence) and the latter uses a map from a barrier (over natural numbers). The precise definition of BQO is given below. First, we define what is a barrier.

Definition 4.4 *Let ω be the least countable ordinal (i.e., set of natural numbers). If $s,t \subseteq \omega$, then $s \le t$ ($s < t$) means that s is a (proper) initial segment of t (as ascending sequences).*

Definition 4.5 *For an infinite set $X \subseteq \omega$, a barrier B on X is a set of finite sets of X s.t. $\phi \notin B$ and*

1. *for every infinite set $Y \subseteq \omega$ there is an $s \in B$ s.t. $s < Y$.*
2. *if $s,t \in B$ and $s \neq t$ then $s \not\subset t$.*

Example 4.6 *Fig. 4.11 illustrates an example of a barrier on natural numbers consisting of* (0), (1), $(2,3),(2,4),(2,5),\cdots$, $(3,4),(3,5),(3,6),(3,7,8),(3,7,9),(3,7,10),\cdots$.

Definition 4.7 *Define $s \lhd t$ to hold if there is an $n > 0$ and $i_0 < \cdots < i_n < \omega$ s.t. for some $m < n$, $s = \{i_0,\cdots,i_m\}$ and $t = \{i_1,\cdots,i_n\}$. (Thus, e.g., $\{3\} \lhd \{5\}$, $\{3,5,6\} \lhd \{5,6,8,9\}$, $\{3,5,6\} \not\lhd \{5,6\}$.)*

Definition 4.8 *Let $\le$ be a QO on Q. If B is a barrier, $f : B \to Q$ is good if there are $s,t \in B$ s.t. $s \lhd t$ and $f(s) \le f(t)$, and f is bad otherwise. f is perfect if for all $s,t \in B$, if $s \lhd t$ then $f(s) \le f(t)$. Q is better-quasi-ordered (BQO) if for every barrier B and every $f : B \to Q$, f is good.*

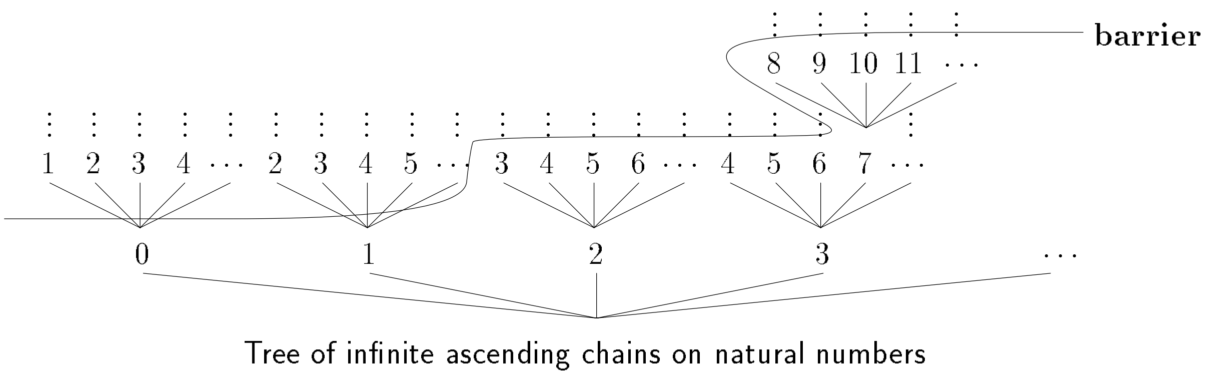


Figure 4.11: An example of barrier

The simple examples of BQO are: a QO over a finite set, well-order (i.e., linear WQO), etc. Further, Kruskal-type theorems guarantee that the extensions of BQOs as the embedding (over various data structures) are again BQOs. Thus, although the definition of BQO is quite complex compared to WQO, their construction is same to one for WQO in practice. Let $P[Q]$ be a power set of Q.

Definition 4.9 *Let $\leq$ be a QO on Q, let α, β be ordinals, and let $Q^\alpha = \{w : \alpha \to Q\}$. For $w \in Q^\alpha$, $v \in Q^\beta$, a one-to-one mapping $\psi : \alpha \to \beta$ is an embedding if $w(n) \leq v(\psi(n))$ for each $n \in \alpha$.*

Theorem 4.10 *[Lav78, NW65] If Q is BQO, $\cup_\alpha Q^\alpha$ is BQO wrt the embedding.*

Corollary 4.11 *[Lav78] If Q is BQO, $(P[Q], \sqsubseteq_1)$ is a BQO.*[2]

Definition 4.12 *Suppose Q is quasi-ordered by $\leq$. A partial ranking on Q is a well-founded (irreflexive) partial order $<'$ on Q s.t. $q <' r$ implies $q < r$. Let B, C be barriers. Then $B \sqsubseteq C$ if*

1. $\cup C \subseteq \cup B$, and

2. for each $c \in C$ there is a $b \in B$ with $b \leq c$.

$B \sqsubset C$ if $B \sqsubseteq C$ and there are $b \in B$, $c \in C$ with $b < c$. For $f : B \to Q$, $g : C \to Q$ and a partial ranking $<'$ on Q, $g \sqsubseteq f$ ($g \sqsubset f$) wrt $<'$ if $B \sqsubseteq C$ ($B \sqsubset C$) and

1. $g(a) = f(a)$ for $a \in B \cap C$,

2. $g(c) <' f(b)$ for $b \in B$, $c \in C$ s.t. $b < c$.

Definition 4.13 *Suppose $<'$ is a partial ranking on Q. For a barrier C, $g : C \to Q$ is minimal bad if g is bad and there is no bad h with $g \sqsubset h$.*

[2]In [Lav78], $\sqsubseteq_1$ is denoted by $\sqsubseteq_m$.

Theorem 4.14 [3] *Let Q be quasi-ordered by $\leq$ and let $<'$ be a partial ranking on Q. Then for any bad f on Q there is minimal bad g s.t. $f \sqsubseteq g$.*

Note that P.Jančar showed that $\sqsubseteq_0$ is WQO if and only if the base domain is ω^2-WQO (i.e., a WQO that not includes Rado's example) [Jan99]. We can also state the fact $\sqsubseteq_0$ is BQO if and only if the base domain is BQO.

Corollary 4.15 *If Q is BQO, $(P[Q], \sqsubseteq_0)$ is a BQO.*

Proof Let $\sqsubset_0 = \sqsubseteq_0 \setminus \sqsupseteq_0$ be a partial ranking. Assume that there exists a minimal bad sequence $f : B \to P[Q]$ for a barrier B. Let $g : B(2) \to f(b_1) \cup f(b_2)$ where $B(2) = \{b_1 \cup b_2 \mid b_1 \triangleleft b_2, b_1, b_2 \in B\}$. Then, $g \sqsubset f$ and the contradiction. ■

This BQO techniques will enlarge the possibility of abstract interpretation, and further investigation for the useful application is desired.

4.4.2 Termination of backward abstract interpretation over infinite abstract domains

Recall that the construction of backward HOMTs in Chapter 3. The basic power domain construction is either $(\sqsubseteq_{-0}, RC)$ (when necessary) or $(\sqsubseteq_{-1}, LC)$ (when sufficient). Then, roughly speaking, the next theorem guarantees that if a (possibly infinite) abstract domain is better-quasi-ordered then a backward abstract interpretation correctly terminates (assuming termination of each primitive function).

Theorem 4.16 *Let a backward HOMT h be $((abs, conc), -, (\sqsubseteq_{-0}, RC), Min)$ (resp. $((abs, conc), -, (\sqsubseteq_{-1}, LC), Max)$) with $abs : P[D] \to P[Abs]$ and $conc : P[Abs] \to P[D]$. Assume that $(abs, conc)$ is necessary (resp. sufficient). If the abstract domain Abs is better-quasi-ordered, then the computation of $h(f)(X)$ terminates for each $X \subseteq Abs$.*

Proof Since an abstract domain is a BQO, its power domain (wrt either $\sqsubseteq_1$ or $\sqsubseteq_0$) is also a BQO by Corollary 4.11 and 4.15. A necessary (resp. sufficient) backward abstract interpretation computes the least fixed point wrt $\sqsubseteq_{-0}$ (resp. $\sqsubseteq_{-1}$), thus from the definition of BQO it converges after finitely many unfolding of the recursive equations. ■

Since any finite domain is better-quasi-ordered, this theorem gives a proper extension of existing termination criteria. Note that this criteria does not hold for a forward abstract interpretation. It is said that a backward abstract interpretation is more efficient than a forward one, and this result actually shows the sharp distinction between them. Unfortunately, we have not found any new applications yet. We hope that further investigation finds interesting examples of analyses/verification over infinite abstract domains.

[3] Theorem 1.9 in [Lav78], or equivalently theorem 9.17 in [Sim85].

Part III

Automatic generation of efficient programs

Chapter 5

Automatic generation based on well-quasi-orders

This chapter demonstrates the generation of a linear time query processing algorithm based on WQO. The target example is a linear time evaluation of a fixed disjunctive monadic query in an indefinite database on a linearly ordered domain, first posed by Van der Meyden [van97]. Van der Meyden showed the existence of a linear time algorithm by Higman's lemma, that is, an evaluation of a fixed disjunctive monadic query in an indefinite database is reduced to the comparison of the database with finitely many minimal models (called *minors*) that are affirmative to the query. Enumerating minors are possible, but enumerating *all* minors is very difficult, since the naive methods cannot decide know when all minors are enumerated. Thus, an actual construction of a linear time evaluation has, until now, not been reported elsewhere.

The obstruction is non-constructive nature of Higman's lemma. Fortunately, Ehrenfeucht et al. showed that *a set L of finite words is regular if and only if L is $\leq$-closed under some monotone well-quasi-order (WQO) $\leq$ over finite words* [EHR83]. This gives the insight that upward closed sets would be described by regular expressions and introducing well-founded orders on such descriptions would give computational contents of Higman's lemma. Murthy and Russell gave a constructive proof by the such scenario [MR90]. We will use their proof, especially their descriptions of (the complement of) upward-closed sets by regular expressions, to decide when all minors are enumerated.

This chapter is organized as follows: Section 5.1 gives the constructive proof of Higman's lemma [MR90] and its extension. Section 5.2 presents the automatic linear time algorithm generation for fixed disjunctive monadic query processing in an indefinite database. Section 5.3 presents the extension of the result in [EHR83] to ω-words. Section 5.4 discusses the possible future direction to reduce the constant by *fold/unfold* program transformation.

5.1 Higman's lemma and the constructive proof

In this section, we will briefly explain the constructive proof of Higman's lemma. Higman's lemma states that any bad sequence has finite length, and the constructive proof of Higman's lemma is presented by constructing the effective well-founded-order (WFO) among bad sequences.

The basic idea is as follows: for a bad sequence, we first assign a union of special regular expressions that approximate the possible choice of the next element to enlarge a bad sequence. Next, we construct an WFO on sets of special regular expressions such that for each bad sequence the regular expression associated with the bad sequence strictly decreases when it is enlarged. Thus, this means that any extension of bad sequences eventually terminates. For details, see [MR90]. We also show an extension of the proof. For definitions related to WQO, please refer Section 4.4 in Chapter 4.

5.1.1 Constructive proof by Murthy-Russell

Lemma 5.1 (Higman's lemma) *[Hig52] If $(\Sigma, \leq)$ is a WQO, then $(\Sigma^*, \preceq)$ is a WQO, where $\preceq$ is a subword relation constructed based on $\leq$ (i.e., $u \preceq v$ if there is an order preserving injection f from u to v s.t. $u_i \leq v_{f(i)}$ for each i).*

The standard proof by Nash-Williams [NW63] is non-constructive, especially the reasoning called *minimal bad sequence*, in which (1) the proof proceeds based on contradictions, (2) the existence of a minimal bad sequence is a result of Zorn's lemma, and (3) the arguments on a minimal bad sequence are heavily impredicative. An example is universal quantification over all bad sequences. A minimal bad sequence is a bad sequence which is minimal wrt the lexicographical order of sizes.

Murthy-Russell, Richman-Stolzenberg, and Coquand-Fridlender independently gave constructive proofs for Higman's lemma [MR90, RS93, CF93][1]. For a constructive proof, we must make the following assumptions.

1. Let A and B be bad sequences of Σ, and let $A \sqsubset_{seq} B$ if, and only if, A is a proper extension of B. $\sqsubset_{seq}$ is well founded and equipped with a well founded induction scheme.

2. The WQO $\leq$ on Σ is decidable.

Classically, the first assumption is obviously based on the WQO property of $\leq$, but, constructively, it is not. The WQO that satisfies the assumptions above is called a *constructive well-quasi-order* (*CWQO*) [Sim88].

We will briefly review the techniques used in [MR90]. We will refer to an empty word as ϵ and an upward closure of words that contains w (i.e., $\{x \in \Sigma^* \mid w \preceq x\}$) as w°.

As a convention, we will refer to the symbols in Σ as $a, b, c, \cdots$, the words in Σ^* as $u, v, w, \cdots$, the finite sequences in Σ as $A, B, \cdots$, the finite sequences in Σ^* as $V, W, \cdots$, the subsets of Σ^* as $L, L', \cdots$, the finite subsets of Σ^* as $\alpha, \beta, \cdots$, the subsets of finite subsets of Σ^* as $\mathcal{L}, \mathcal{L}', \cdots$, the special periodic expressions called *sequential regular expressions* as $\sigma, \theta, \cdots$, the finite sets of sequential regular expressions as $\Theta, \Theta_1, \Theta_2, \cdots$, the special power set expressions called *base expressions* as $\bar{\sigma}, \bar{\theta}, \cdots$, and the finite sets of base expressions as $\Phi, \Phi_1, \Phi_2, \cdots$.

[1] A similar idea to [RS93] is also found in [Sim88].

Definition 5.2 *Let $b \in \Sigma$, and let $A = a_1, a_2, \cdots, a_k$ be a bad sequence in Σ. The constant expression $(b - A)$ denotes a subset of Σ defined by*

$$\{x \in \Sigma \mid b \leq x \ \wedge \ a_i \not\leq x \text{ for each } \ i \leq k\},$$

and the starred expression $(\Sigma - A)^*$ denotes a subset of Σ^* defined by

$$\{w = c_1 c_2 \cdots c_n \in \Sigma^* \mid a_j \not\leq c_i \text{ for each } i \leq n, j \leq k\}.$$

The concatenation of A and $a \in \Sigma$ is $A|a$.

A sequential regular expression (sequential r.e.) σ is a (possibly empty) concatenation of either constant or starred expressions. The size $size(\sigma)$ of σ is the number of the concatenation. For a finite set Θ of sequential expressions, we define $L(\Theta) = \cup_{\sigma \in \Theta} \sigma$.

Let $w_1, w_2, w_3, \cdots$ be a bad sequence of elements in Σ^*. We will explicitly construct a finite set Θ_k of sequential r.e.'s for $w_1, w_2, \cdots, w_k$ such that $\Sigma^* \setminus (w_1^\circ \cup \cdots \cup w_k^\circ) \subseteq L(\Theta_k)$. For describing Θ_k, we define $\Theta(\sigma, w)$. The basic idea of $\Theta(\sigma, w)$ is that, for a word not to be a superword of w, it can only contain a proper subword of w. So what we do is write down the sequential r.e.'s that accept classes of words containing different proper subwords of w.

Definition 5.3 *For sequential r.e.'s $\sigma_1, \cdots, \sigma_n$, we define their concatenation $\sigma_1 \cdots \sigma_n$ as $\{w_1 \cdots w_n \mid w_i \in \sigma_i \text{ for } i \leq n\}$, and denote $+$ for the union operation. Let σ be a sequential r.e. and let $w \in \sigma$. We will define $\Theta(\sigma, w)$ as follows.*

1. *When σ is a constant expression $(b - A)$*[2]*, we can identify w as a single symbol in Σ because $\sigma \subseteq \Sigma$. Then $\Theta(\sigma, w) = (b - A|w) + \epsilon$.*

2. *When σ is a starred expression $(\Sigma - A)^*$*[3]*, let $w = c_1 c_2 \cdots c_l$ with $c_j \in \Sigma$ for each j. Then $\Theta(\sigma, w)$ is*

$$\cup_{j=1}^{l} \left\{ \begin{array}{l} (\Sigma - A|c_1)^*((c_1 - A) + \epsilon) \cdots (\Sigma - A|c_{j-1})^*((c_{j-1} - A) + \epsilon) \\ (\Sigma - A|c_j)^*((c_{j+1} - A) + \epsilon)(\Sigma - A|c_{j+1})^* \cdots ((c_l - A) + \epsilon)(\Sigma - A|c_l)^* \end{array} \right\}$$

3. *When $\sigma = \sigma_1 \sigma_2 \cdots \sigma_n$, where σ_i is either a constant or starred expression, we fix a decomposition of w into σ_is (i.e., $w = w_1 w_2 \cdots w_n$) with $w_i \in \sigma_i$ for each $i \leq n$. Then*

$$\Theta(\sigma, w) = \cup_{i=1}^{n} \{\sigma_1 \cdots \sigma_{i-1} \theta \sigma_{i+1} \cdots \sigma_n \mid \theta \in \Theta(\sigma_i, w_i)\}.$$

Let Θ be a finite set of sequential r.e.'s. Let $size(\Theta)$ be $max(\{size(\sigma) \mid \sigma \in \Theta\})$. The following lemma shows that if we remove the sequential r.e. σ from Θ, and replace it with the set $\Theta(\sigma, w)$ with $w \in \sigma$, the resulting (finite) set of sequential r.e.'s includes all the finite words in $L(\Theta)$ not containing w.

[2]Ref. [MR90] has a flaw that $\Theta(\sigma, w)$ is simply defined as $(b - A|w)$.

[3]Ref. [MR90] has a flaw that $\Theta(\sigma, w)$ is simply defined as $(\Sigma - A|c_1)^*(c_1 + \epsilon) \cdots (c_{l-1} + \epsilon)(\Sigma - A|c_l)^*$.

Lemma 5.4 *Let $L \subseteq \Sigma^*$. Assume that there is a finite set Θ of sequential r.e.'s such that $L \subseteq L(\Theta)$. For any $w \in L$, $\sigma \in \Theta$ with $w \in \sigma$,*

$$L \setminus w^\circ \subseteq L((\Theta \setminus \{\sigma\}) \cup \Theta(\sigma, w)).$$

Thus, for a bad sequence $w_1, w_2, \cdots$, we can construct Θ_k by starting from Σ^* and repeating the applications of Lemma 5.4. If this process terminates, Θ_k eventually empties. This means that $\preceq$ is a WQO. For termination, we construct a well-founded order $\sqsubset_{setexp}$, which strictly decreases when Lemma 5.4 is applied. This gives a constructive proof of Higman's lemma.

Definition 5.5 *For the finite sequences A, B in Σ, $A \sqsubset_{seq} B$ if B is a proper prefix of A. For any pair of constant expressions $(a - A)$ and $(b - B)$, $(a - A) \sqsubset_{const} (b - B)$ if $a = b \wedge A \sqsubset_{seq} B$. For any pair of starred expressions $(\Sigma - A)^*$ and $(\Sigma - B)^*$, $(\Sigma - A)^* \sqsubset_* (\Sigma - B)^*$ if $A \sqsubset_{seq} B$. Let $\sqsubset_{exp} = \sqsubset_{const} \cup \sqsubset_* \cup \{(a - A)\} \times \{(\Sigma - B)\}$ (i.e., all the constant expressions are below the starred expressions). Let $\sqsubset_{setexp}$ be a multiset extension [NZ79] of $\sqsubset_{exp}$.*

We define an ordering $\sqsubset_{re}$ of sequential r.e.'s by $\sigma \sqsubset_{re} \theta \Leftrightarrow \biguplus_{i=1}^{k}\{\sigma_i\} \sqsubset_{setexp} \biguplus_{j=1}^{l}\{\theta_j\}$, for $\sigma = \sigma_1 \cdots \sigma_k$ and $\theta = \theta_1 \cdots \theta_l$, where the σ_is and θ_js are either constant or starred expressions. We also denote a multiset extension of $\sqsubset_{re}$ with $\sqsubset_{setre}$.

Theorem 5.6 *Let $W = w_1, w_2, \cdots, w_k$ be a finite bad sequence in Σ^*. One can effectively compute a finite set Θ_i of sequential r.e.'s for $i \leq k$ such that*

$$\Sigma^* \setminus (w_1^\circ \cup w_2^\circ \cup \cdots \cup w_i^\circ) \subseteq L(\Theta_i)$$

and $\Theta_{i+1} \sqsubset_{setre} \Theta_i$ for $i < k$.

Corollary 5.7 *If $(\Sigma, \leq)$ is a CWQO, then $(\Sigma^*, \preceq)$ is a CWQO.*

Example 5.8 *Let $\Sigma = \{a, b\}$. Consider the bad sequence $ab, bbaa, ba, bb, a, b$ wrt $\preceq$.*

$$\begin{array}{rcl}
\Theta_0 & = & (\Sigma - \epsilon)^* \\
\Theta_1 & = & (\Sigma - a)^*(b + \epsilon)(\Sigma - b)^* \cup (\Sigma - a)^*(a + \epsilon)(\Sigma - b)^* \\
 & = & \{b^*a^*\} \\
\Theta_2 & = & (b + \epsilon)(\Sigma - b) \cup (\Sigma - a)(a + \epsilon) \\
 & = & \{ba^*, b^*a\} \\
\Theta_3 & = & (\Sigma - b) \cup (b + \epsilon) \cup (a) \cup (\Sigma - a) \\
 & = & \{a^*, b^*\} \\
\Theta_4 & = & (\Sigma - b) \cup (b + \epsilon) \\
 & = & \{a^*, b\} \\
\Theta_5 & = & \{\epsilon, b\} \\
\Theta_6 & = & \{\epsilon\}
\end{array}$$

5.1.2 An Extension

For our purposes, we need further extension to the sets of finite sets of finite words (which is not included in [MR90]). Let $\mathcal{F}(\Sigma^*)$ be the set of all finite sets of Σ^*. Assume that $(\Sigma, \leq)$ satisfies the CWQO assumptions. Note that an embedding $(\Sigma^*, \preceq)$ satisfies them as well. We define $\alpha \leq_m \beta$ for $\alpha, \beta \in \mathcal{F}(\Sigma^*)$ if, for each $x \in \alpha$, there exists $y \in \beta$ such that $x \leq y$. We also denote the upward closure of α in $\mathcal{F}(\Sigma^*)$ (i.e., $\{\gamma \in \mathcal{F}(\Sigma^*) \mid \alpha \leq_m \gamma\}$) with α°.

Definition 5.9 *Let $W = w_1, w_2, \cdots, w_k$ be a finite bad sequence in Σ^*. The base expression is*

$$(\Sigma^* \ominus W) = \mathcal{F}(\{u \in \Sigma^* \mid w_i \not\leq u \text{ for each } \ i \leq k\})$$

We define $\Sigma^* \ominus V \sqsubset_{base} \Sigma^* \ominus W$ if $V \sqsubset_{seq} W$. For a finite set Φ of base expressions, we define $\mathcal{L}(\Phi) = \cup_{\bar{\sigma} \in \Phi} \bar{\sigma}$.

Let $\alpha_1, \alpha_2, \cdots$ be a bad sequence of elements in $\mathcal{F}(\Sigma^*)$. We will explicitly construct a finite set Φ_k of base expressions for $\alpha_1, \cdots, \alpha_k$ such that $\mathcal{F}(\Sigma^*) \setminus (\alpha_1^\circ \cup \cdots \cup \alpha_k^\circ) \subseteq \mathcal{L}(\Phi_i)$. For describing Φ_k, we define $\Phi(\Sigma^* \ominus V, \alpha)$. The basic idea of $\Phi(\Sigma^* \ominus V, \alpha)$ is, for a finite set not to be a superset of α, it must not contain at least one of the elements in α. What we do is write down base expressions which accept finite sets not containing some element of α.

Definition 5.10 *Let $(\Sigma^* \ominus V)$ be the base expression for a finite bad sequence V in Σ^*, and let $\alpha \in (\Sigma^* \ominus V)$. We then define $\Phi(\Sigma^* \ominus V, \alpha) = \{\Sigma^* \ominus V|v \mid v \in \alpha \ \wedge \ v_i \not\leq v \text{ for each } v_i \in V\}$.*

Lemma 5.11 *Let $\mathcal{L} \subseteq \mathcal{F}(\Sigma^*)$. Assume that there is a finite set Φ of base expressions such that $\mathcal{L} \subseteq \cup_{\bar{\sigma} \in \Phi} \bar{\sigma}$. For any $\alpha \in L$ and $\bar{\sigma} \in \Phi$ with $\alpha \in \bar{\sigma}$,*

$$\mathcal{L} \setminus \alpha^\circ \subseteq \mathcal{L}((\Phi \setminus \{\bar{\sigma}\}) \cup \Phi(\bar{\sigma}, \alpha)).$$

Let $\sqsubset_{base}$ be the multiset extension of $\sqsubset_{seq}$. Since $\preceq$ is a WQO on Σ^*, $\sqsubset_{seq}$ is well founded, and so is $\sqsubset_{base}$. Thus, by using a constructive proof similar to Higman's lemma, we obtain the next theorem.

Theorem 5.12 *Let $\mathcal{A} = \alpha_1, \alpha_2, \cdots, \alpha_k$ be a finite bad sequence in $\mathcal{F}(\Sigma^*)$. Then one can effectively compute a finite set Φ_i of base expressions for $i \leq k$ such that*

$$\mathcal{F}(\Sigma^*) \setminus (\alpha_1^\circ \cup \alpha_2^\circ \cup \cdots \cup \alpha_i^\circ) \subseteq \mathcal{L}(\Phi_i)$$

and $\Phi_{i+1} \sqsubset_{base} \Phi_i$ for $i < k$.

Corollary 5.13 *If $(\Sigma^*, \leq)$ is a CWQO, then $(\mathcal{F}(\Sigma^*), \leq_m)$ is a CWQO.*

Example 5.14 *Let $\Sigma = \{a, b\}$. Consider the bad sequence $\{ab, bbaa\}, \{ba, bb\}, \{a, b\}$ wrt $\leq_m$.*

$$\begin{array}{rcl}
\Phi_0 & = & \{(\Sigma^* \ominus \epsilon)\} \ = \ \{\mathcal{F}(\Sigma^*)\} \\
\Phi_1 & = & \{(\Sigma^* \ominus (ab)),\ (\Sigma^* \ominus (bbaa))\} \\
 & = & \{\mathcal{F}(\{b^*a^*\}),\ \mathcal{F}(\{a^*(b+\epsilon)a^*b^*(a+\epsilon)b^*\})\} \\
\Phi_2 & = & \{(\Sigma^* \ominus (ab, ba)),\ (\Sigma^* \ominus (ab, bb)), \\
 & & (\Sigma^* \ominus (bbaa, ab)),\ (\Sigma^* \ominus (bbaa, bb))\} \\
 & = & \{\mathcal{F}(\{a^*, b^*\}),\ \mathcal{F}(\{(b+\epsilon)a^*, b^*(a+\epsilon)\}), \\
 & & \mathcal{F}(\{a^*(b+\epsilon)a^*\})\} \\
\Phi_3 & = & \{(\Sigma^* \ominus (ab, ba, a)),\ (\Sigma^* \ominus (ab, ba, b)), \\
 & & (\Sigma^* \ominus (ab, bb, a)),\ (\Sigma^* \ominus (ab, bb, b)), \\
 & & (\Sigma^* \ominus (bbaa, ab, a)),\ (\Sigma^* \ominus (bbaa, ab, b)), \\
 & & (\Sigma^* \ominus (bbaa, bb, a)),\ (\Sigma^* \ominus (bbaa, bb, b))\} \\
 & = & \{\mathcal{F}(\{a^*\}),\ \mathcal{F}(\{b^*\})\}
\end{array}$$

By combining the techniques in Section 5.1.1, we can write down each basic expression as a finite set of sequential r.e.'s. Thus, $\mathcal{F}(\Sigma^*) \setminus (\alpha_1^\circ \cup \alpha_2^\circ \cup \cdots \cup \alpha_i^\circ)$ is, too. For instance, $\Phi_1 = \mathcal{F}(\Sigma^*) \setminus \{ab, bbaa\}^\circ$ in Example 5.14 can be written as follows:

$$\begin{array}{rcl}
\Phi_1 & = & \{(\Sigma^* \ominus (ab)),\ (\Sigma^* \ominus (bbaa))\} \\
 & = & \{\mathcal{F}((\Sigma - a)^*(b+\epsilon)(\Sigma - b)^* \cup (\Sigma - a)^*(a+\epsilon)(\Sigma - b)^*), \\
 & & \mathcal{F}((\Sigma - b)^*(b+\epsilon)(\Sigma - b)^*(a+\epsilon)(\Sigma - a)^*(a+\epsilon)(\Sigma - a)^* \cup \\
 & & \quad (\Sigma - b)^*(b+\epsilon)(\Sigma - b)^*(b+\epsilon)(\Sigma - a)^*(a+\epsilon)(\Sigma - a)^*)\}
\end{array}$$

5.2 Algorithm generation based on WQO techniques

Throughout this section, we use the symbol D for an indefinite database, and fix a disjunctive monadic query $\varphi = \psi_1 \vee \psi_2 \vee \cdots \vee \psi_m$, where the ψ_i's are conjunctive components (i.e., conjunctive monadic queries). In our situation, combining Theorem 5.6 and 5.12 shows that $\Sigma^* \ominus W$ is approximated with a finite set of sequential r.e.'s. More precisely,

$$(\Sigma^* \ominus W) \subseteq \mathcal{F}(\Theta(\cdots \Theta(\Theta(\Sigma^*, w_1), w_2), \cdots, w_k)$$

for a finite bad sequence $W = w_1, w_2, \cdots, w_k$ in Σ^*, and

$$\Phi(\Sigma^* \ominus W, \alpha) = \{\mathcal{F}(\Theta(\Theta \cdots \Theta(\Sigma^*, w_1), \cdots, w_k), v) \mid v \in \alpha \ \wedge \ w_i \not\leq v\}.$$

From now on, we use a finite set of finite sets of sequential r.e.'s as a substitute for a base expression Φ.

5.2.1 Disjunctive monadic query on indefinite database

As a target example, we used the linear time fixed disjunctive monadic query processing of indefinite database proposed by Van der Meyden [van97]. He posed the following unsolved problem:

In a fixed disjunctive monadic query, there is an algorithm answering the query, which is linear wrt the size of the indefinite database on a linearly ordered domain. What is the actual algorithm?

Here, we briefly review his results. For details, please refer to [van97].

Proper atoms are of the form $P(a)$, where P is a predicate symbol, and a is a tuple of constants or variables. *Order atoms* are of the form $u < v$, where u and v are order constants or variables. An *indefinite database* D is a set of ground atoms. The atoms are either proper atoms or order atoms. A model $\mathcal{D}$ of D is a linearly ordered domain (such as time) satisfying D. D is a collection of partial facts on a linearly ordered domain, and thus is referred to indefinite.

We concentrate on *monadic query processing*, (i.e., the database and queries contain only monadic predicate symbols except for $<$). A predicate symbol is *monadic* if its arity is less than or equal to one. The class of monadic queries is restrictive, but contains nontrivial problems, such as a comparison between two gene alignments (regarding C, G, A, T as monadic predicates).

For linearly ordered domains $\mathcal{D}$ and $\mathcal{D}'$, a map $f : \mathcal{D} \to \mathcal{D}'$ is order-preserving if any constant $t, t' \in \mathcal{D}$ with $t < t'$ holds $f(t) < f(t')$, and predicate-preserving if a proper atom $P(t) \in \mathcal{D}$ implies $P(f(t)) \in \mathcal{D}'$ and vice versa. An embedding e is an order-preserving, predicate-preserving one-to-one map, and a projection p is an order-preserving, predicate-preserving onto map. A model $\mathcal{D}$ is embedded to a model $\mathcal{D}'$ if there is an embedding e from $\mathcal{D}$ to $\mathcal{D}'$, and A model $\mathcal{D}'$ is projected to a model $\mathcal{D}$ if there is a projection p from $\mathcal{D}'$ to $\mathcal{D}$.

A *disjunctive query* (or, simply a *query*) is a positive existential first-order clause constructed from proper and order atoms using only $\exists$, $\wedge$, and $\vee$. A *conjunctive query* is a first-order clause constructed from proper atoms and order atoms using only $\exists$ and $\wedge$. For simplicity, queries are expressed in disjunctive normal forms, i.e., disjunctions of conjunctive queries. Each conjunctive query in a disjunctive normal form is called a *conjunctive component*.

For an indefinite database D and a query φ, we define $D \models \varphi$ if φ is valid in any model of D. For instance, let $D = \{P(a), Q(b), a < b, Q(c), R(d), c < d, R(e), P(f), e < f\}$, and let $\varphi = \psi_1 \vee \psi_2 \vee \psi_3$ where

$$\begin{cases} \psi_1 = \exists xyz[P(x) \wedge Q(y) \wedge R(z) \wedge \ x < y < z], \\ \psi_2 = \exists xyz[Q(x) \wedge R(y) \wedge P(z) \wedge \ x < y < z], \text{ and} \\ \psi_3 = \exists xyz[R(x) \wedge P(y) \wedge Q(z) \wedge \ x < y < z]. \end{cases}$$

As a result, $D \models \varphi$. Note that neither $D \models \psi_1$, $D \models \psi_2$, nor $D \models \psi_3$.

Definition 5.15 *A conjunctive query is sequential if its form is*

$$\exists t_1 t_2 \cdots t_n \ [P_1(t_1) \wedge \cdots \wedge P_n(t_n) \wedge \ t_1 < t_2 < \cdots < t_n].$$

Let $Pred$ be a set of monadic predicates, and let $\Sigma = \mathcal{P}(Pred)$ be the power set of $Pred$. Σ^* is the set of all the finite words of the symbols in Σ. Without losing generality, we can assume that a monadic query does not contain constants, i.e., if the query φ contains the constant u, we can eliminate u by adding $P_u(u)$ to a database and replacing φ with $\exists x\ [P_u(x) \wedge \varphi]$ for a new predicate P_u. Thus, up to variable-renaming, sequential monadic queries correspond one-to-one to words in Σ^*. For instance, $\exists t_1t_2t_3\ [P(t_1) \wedge Q(t_1) \wedge P(t_2) \wedge R(t_3) \wedge\ t_1 < t_2 < t_3]$ corresponds to $\{P, Q\}\{P\}\{R\}$. This correspondence is naturally extended to conjunctive queries, i.e., correspondence from a conjunctive query to a finite set of words in $\mathcal{F}(\Sigma^*)$. For instance, $\exists t_1t_2t_3\ [P(t_1) \wedge Q(t_1) \wedge P(t_2) \wedge R(t_3) \wedge\ t_1 < t_2 < t_3]\ \wedge\ \exists t_1t_2t_4\ [P(t_1) \wedge Q(t_1) \wedge P(t_2) \wedge S(t_4) \wedge\ t_1 < t_2 < t_4]$ corresponds to $\{\ \{P, Q\}\{P\}\{R\},\ \{P, Q\}\{P\}\{S\}\ \}$. If ψ is a conjunctive monadic query, a *path* in ψ is a maximal (wrt implication) sequential subquery of ψ. We use the expression $Paths(\psi)$ for the subset of Σ^* corresponding to paths of ψ, $length(\psi)$ for the sum of the lengths of all the paths, and $base(\psi)$ for the set of all predicate symbols appearing in ψ.

Lemma 5.16 *Let D be a monadic database and ψ be a conjunctive monadic query. Then, $D \models \psi$ if and only if $D \models p$ for every path $p \in Paths(\psi)$.*

Let $P_1, P_2, \cdots, P_n$ be either proper or order atoms. By regarding the indefinite database $D = \{P_1, P_2, \cdots, P_n\}$ as a conjunctive monadic formula $P_1 \wedge P_2 \wedge \cdots \wedge P_n$, the paths of the database are similarly defined. We denote the set of paths as $Paths(D)$. Note that the paths in an indefinite database can be computed in linear time wrt the size of the database.

Lemma 5.17 *Let ψ be a sequential query, and let $\preceq$ be a subword relation on Σ^* constructed from $\subseteq$ on Σ (i.e., $u \preceq v$ if there is an order preserving injection f from u to v s.t. $u_i \subseteq v_{f(i)}$ for each i). Then $D \models \psi$ if, and only if, there is a path $\psi' \in Paths(D)$ s.t. $\psi \preceq \psi'$.*

For a disjunctive query φ, $D \models \varphi$ may be *true* even if $D \models \psi$ does not for each conjunctive component ψ of φ. This makes it difficult to judge whether $D \models \varphi$. For the indefinite databases, D_1 and D_2, $D_1 \sqsubseteq D_2$ if $Paths(D_1) \leq_m Paths(D_2)$, where $U \leq_m V$ if $\forall u \in U \exists v \in V$ s.t. $u \preceq v$. We frequently identify an indefinite database D and the set of its paths $Paths(D)$, and also identify $\sqsubseteq$ and $\leq_m$.

Theorem 5.18 *For any disjunctive monadic query φ, if $D_1 \models \varphi$ and $D_1 \sqsubseteq D_2$, then $D_2 \models \varphi$.*

Here, we remark on the existence of the linear time algorithm to decide whether $D \models \varphi$ for a fixed disjunctive query φ. Elements in the $Pred$ of interest are elements in the monadic queries. Thus, without loss of generality, we can assume that $Pred$ is finite, and the set inclusion $\subseteq$ in $\Sigma = \mathcal{P}(Pred)$ is a WQO. Then, according to Higman's lemma, $(\Sigma^*, \preceq)$ and $(\mathcal{F}(\Sigma^*), \leq_m)$ are WQOs, where Σ^* is the set of finite words of Σ and $\mathcal{F}(\Sigma^*)$ is the set of finite sets of Σ^*. Based on Theorem 5.18, the set of indefinite databases which hold a fixed disjunctive query φ is upward closed wrt $\sqsubseteq$. Thus the problem of judging whether $D \models \varphi$ is reduced to a comparison of D with minimal (wrt $\sqsubseteq$) indefinite databases $\{D_i\}$ with $D_i \models \varphi$, called *minors*. The judgment can be made linearly in the size of D. From this observation, the next theorem follows.

Theorem 5.19 *Let us fix a disjunctive monadic query φ. Then, there exists a linear time algorithm to decide $D \models \varphi$ for a monadic database D.*

Example 5.20 *Consider the example above. Recall that* $\varphi = \psi_1 \vee \psi_2 \vee \psi_3$ *where*

$$\begin{cases} \psi_1 &= \exists xyz[P(x) \wedge Q(y) \wedge R(z) \wedge \; x < y < z], \\ \psi_2 &= \exists xyz[Q(x) \wedge R(y) \wedge P(z) \wedge \; x < y < z], \; \textit{and} \\ \psi_3 &= \exists xyz[R(x) \wedge P(y) \wedge Q(z) \wedge \; x < y < z]. \end{cases}$$

Then, $D \models \varphi$ *if and only if there exists* D' *with* $D' \sqsubseteq D$ *such that* $Paths(D')$ *is one of*

$$\begin{array}{l} \{\ \{P\}\{Q\},\ \{Q\}\{R\},\ \{R\}\{P\}\ \},\ \{\ \{P\}\{Q\}\{R\}\ \},\ \{\ \{Q\}\{R\}\{P\}\ \},\ \{\ \{R\}\{P\}\{Q\}\ \}, \\ \{\ \{P\}\{Q\}\{P\},\ \{Q\}\{R\}\ \},\ \{\ \{Q\}\{R\}\{Q\},\ \{R\}\{P\}\ \},\ \{\ \{R\}\{P\}\{R\},\ \{P\}\{Q\}\ \}. \\ \{\ \{P\}\{R\}\{P\},\ \{Q\}\{R\}\ \},\ \{\ \{Q\}\{P\}\{Q\},\ \{R\}\{P\}\ \},\ \{\ \{R\}\{Q\}\{R\},\ \{P\}\{Q\}\ \}. \end{array}$$

Note that if a disjunctive monadic query varies, the complexity becomes co-NP. This theorem only states the existence of a linear time algorithm, and the construction, which is reduced to the generation of all the minimal indefinite databases wrt $\sqsubseteq$, will be shown below.

5.2.2 Design of disjunctive query processing algorithm

We say *minors* for minimal indefinite databases wrt $\sqsubseteq$ that are valid for φ, and a set of all minors is denoted by $\mathcal{M}_\varphi$. From the observation in Section 3, we know that the essence of linear time algorithm generation for deciding $D \models \varphi$ is reduced to generating $\mathcal{M}_\varphi$. Thus, our aim is to generate $\mathcal{M}_\varphi$.

Let *Pred* be the set of monadic predicate symbols appearing in φ, and let $\Sigma = \mathcal{P}(Pred)$. Σ is a lattice wrt set inclusion $\subseteq$, and $\subseteq$ is a WQO because *Pred* is finite. Thus, from Corollary 5.7 and 5.13, $\leq_m$ on $\mathcal{F}(\Sigma^*)$ is a WQO. Then the *ideal* algorithm to generate minors, which is presented in Fig. 5.1, has the following predicates and functions.

- `Enumerate(n)`. Enumerates all elements of $\mathcal{F}(\Sigma^*)$ (i.e., an one-to-one onto map from $\mathbf{N}$ to $\mathcal{F}(\Sigma^*)$ satisfying that `Enumerate(i)` $\leq_m$ `Enumerate(j)` implies $i \leq j$).
- `Exclude(L,`α`)`. Compute the subset of `L` consisting of elements *not* greater-than-equal to α wrt $\leq_m$.
- `QueryTest(`α`)`. For $\alpha \in \mathcal{F}(\Sigma^*)$, decide whether $D \models \alpha$ implies $D \models \varphi$.
- `In(x,L)`. Decide whether an element `x` is in `L`.
- `ExistsMinor(`Φ`)`. For a finite set Φ of finite sets of sequential r.e.'s, decide whether there exists $\alpha \in \cup_{\Theta \in \Phi} \mathcal{F}(L(\Theta))$ satisfying `QueryTest(`α`)`.

We choose `Enumerate(n)` as any enumeration with the condition above. The implementation of `QueryTest(`α`)` is as follows.

`QueryTest(`α`)` is decidable, because this is specified in the monadic second order logic S1S [Tho90a]. To illustrate, let $\varphi = \psi_1 \vee \psi_2 \vee \psi_3$, where

$$\begin{cases} \psi_1 &= \exists xyz[P(x) \wedge Q(y) \wedge R(z) \wedge \; x < y < z], \\ \psi_2 &= \exists xyz[Q(x) \wedge R(y) \wedge P(z) \wedge \; x < y < z], \; \text{and} \\ \psi_3 &= \exists xyz[R(x) \wedge P(y) \wedge Q(z) \wedge \; x < y < z]. \end{cases}$$

and let $\alpha = \{PQ, QR, RP\}$. `QueryTest(`α`)` is represented in S1S as

```
begin
  M:={ };
  L:={{Σ*}};
  n=0;
  begin
    while ExistsMinor(L) do
      begin
        NotFound:= true;
        while NotFound do
        begin
          if QueryTest(Enumerate(n)) and In(Enumerate(n),L) then
            begin
              add Enumerate(n) to M;
              L:= Exclude(L,Enumerate(n));
              NotFound:= false;
            end
          n:= n+1;
        end
      end
    return M;
  end
end
```

Figure 5.1: The ideal algorithm to detect minors $\mathcal{M}_\varphi$

$$\begin{array}{rl}(\exists xyzuvw. & P(x) \wedge Q(y) \wedge x < y \ \wedge Q(z) \wedge R(u) \wedge z < u \ \wedge R(v) \wedge P(w) \wedge v < w \\ & \wedge\ x \neq y \ \wedge\ x \neq z \ \wedge\ x \neq u \ \wedge\ x \neq v \ \wedge\ x \neq w \\ & \wedge\ y \neq z \ \wedge\ y \neq u \ \wedge\ y \neq v \ \wedge\ y \neq w \ \wedge\ z \neq u \\ & \wedge\ z \neq v \ \wedge\ z \neq w \ \wedge\ u \neq v \ \wedge\ u \neq w \ \wedge\ v \neq w) \\ \to & (\psi_1 \vee \psi_2 \vee \psi_3)\end{array} .$$

This is valid and `QueryTest`(α) is *true*.

The difficulties are at `Exclude(L,`α`)` and `ExistsMinor(`Φ`)`. For `Exclude(L,`α`)`, we will use the approximation constructed `ApproxExclude(L,`α`)` by the repeated applications of Theorem 5.6 and Theorem 5.12, and we need some modification of the algorithm in Fig. 5.1. `In(`α`,L)` will be also given in this context. They will be formally shown in Section 5.2.3.

The test of `ExistsMinor(L)` ensures termination of the algorithm, and its construction will be shown in Section 5.2.4. Note that if `ExistsMinor(L)` then eventually `QueryTest(Enumerate(n)) and In(Enumerate(n),L)` becomes *true*.

```
1:    begin
2:      M:={ };
3:      $\Phi$:={{$\Sigma^*$}};
4:      n=0;
5:      begin
6:        while ExistsMinor($\Phi$) do
7:          begin
8:            NotFound:= true;
9:            while NotFound do
10:           begin
11:             if QueryTest(Enumerate(n)) and In(Enumerate(n),$\Phi$) then
12:               begin
13:                 add Enumerate(n) to M;
14:                 $\Phi$:= ApproxExclude($\Phi$,Enumerate(n));
15:                 NotFound:= false;
16:               end
17:             n:= n+1;
18:           end
19:         end
20:       M:= Minimize(M);
21:       return M;
22:     end
23:   end
```

Figure 5.2: The revised algorithm to detect minors $\mathcal{M}_\varphi$

5.2.3 Implementation of disjunctive query processing algorithm

Precise computation of `Exclude(L,`α`)` is not easy. Instead, by using the regular expression-like construction of a base expression Φ in Section 5.1, we approximate it as `ApproxExclude(`Φ`,`α`)` with

$$\texttt{Exclude(L,}\alpha\texttt{)} \subseteq \texttt{ApproxExclude(}\Phi\texttt{,}\alpha\texttt{)}$$

for `L` $\subseteq \mathcal{L}(\Phi)$. Corresponding to this setting, the algorithm presented in Fig. 5.1 is modified as in Fig. 5.2. The modifications are at L14 and L20 (in *italic*); since `Exclude(`$\mathcal{L}(\Phi),\alpha$`))` $\subseteq$ `ApproxExclude(`Φ`,`α`)`, there may be some element β such that $\beta \geq_m \alpha$ for some minor α and β eventually satisfies the condition in L11. To remove such garbage, we need to minimize the final result by `Minimize(M)` at L20.

- `ApproxExclude(`Φ, α`)`. For a finite set Φ of finite sets of sequential r.e.'s and $\alpha \in \mathcal{F}(\Sigma^*)$, construct a finite set Φ' of finite sets of sequential r.e.'s such that $\cup_{\Theta \in \Phi} L(\mathcal{F}(\Theta)) \setminus \alpha^\circ \subseteq \cup_{\Theta' \in \Phi'} L(\mathcal{F}(\Theta'))$ and $\Phi >_m \Phi'$.

- `Minimize(M)`. For a finite subset M of $\mathcal{F}(\Sigma^*)$, minimize `M` wrt $\leq_m$.

`Exclude` is implemented by repeating the applications of Theorem 5.6 and Theorem 5.12. `In(`α`,`Φ`)` is computed by checking whether each element in α is contained one of sequential r.e.'s in $\Theta(\in \Phi)$. `Minimize` is easily computed by using $\leq_m$.

5.2.4 Construction of `ExistsMinor(L)`

We will construct the *upper bound* of indefinite database for `ExistsMinor(`Φ`)` for a base expression Φ. The basic idea is to construct database $D_{\theta,n}$ for a sequential r.e. $\theta = \sigma_1 \cdots \sigma_l$ (where σ_i is a constant or starred expressions) such that

$$\begin{aligned} Paths(D_{\theta,n}) &= \psi(\sigma_1)^{n_1} \times \cdots \times \psi(\sigma_l)^{n_l} \\ &= \{P_{1,1} \cdots P_{1,n_1} \cdots P_{l,n_l} \mid P_{i,j} \in \psi(\sigma_i) \text{ for } 1 \leq i \leq l, 1 \leq j \leq n_i\} \end{aligned}$$

where $n_i = 1$ if σ_i is a constant expression, and $n_i = n$ otherwise. Then we will show that $D_{\theta,n} \models \varphi$ and $D_{\theta,n'} \models \varphi$ are equivalent for sufficiently large n, n'. Then, by computing the upper bound n for each set $\Theta(\in \Phi)$ of sequential r.e.'s. `ExistsMinor(`Φ`)` becomes equivalent to $\cup_{\theta \in \Phi} D_{\theta,n} \models \varphi$.

Definition 5.21 *Let A be a bad sequence $a_1, a_2, \cdots, a_k$ in Σ and $b \in \Sigma$. For the constant expression $(b - A)$ and the starred expression $(\Sigma - A)^*$, we define $\psi((b - A))$ as the set of the maximum elements in $(b - A)$, i.e.,*

$$max(\{X \subseteq base(\varphi) \mid b \subseteq X, a \not\subseteq X \text{ for each } a \in A\})$$

$\psi((\Sigma - A)^*)$ as the set of the maximum element in $\Sigma \setminus (a_1^\circ \cup a_2^\circ \cup \cdots a_k^\circ)$, i.e.,

$$max(\{X \subseteq base(\varphi) \mid a \not\subseteq X \text{ for each } a \in A\})$$

For either constant or starred expression σ, $\psi(\sigma)$ can be computed, because $\Sigma = base(\varphi)$ (i.e., the set of all subsets of predicate symbols appearing in the fixed query, see Section 5.2.1) is finite in our context.

Definition 5.22 *For the sequential r.e. $\theta = \sigma_1 \cdots \sigma_l$, we define an indefinite database $D_{\theta,n}$ by*

$$\wedge_{P_{i,j} \in \psi(\sigma_i), 1 \leq j \leq n_i} \exists x_{1,1} \cdots x_{l,n_l}[P_{1,1}(x_{1,1}) \wedge \cdots \wedge P_{l,n_l}(x_{l,n_l}) \wedge x_{1,1} < \cdots < x_{l,n_l}]$$

where $n_i = 1$ if σ_i is a constant expression, and $n_i = n$ otherwise.

Definition 5.23 *Let Θ be a finite set of sequential r.e.'s $\theta_1, \cdots, \theta_s$. Let $\varphi = \psi_1 \vee \psi_2 \vee \cdots \vee \psi_m$ where $\psi_1, \cdots, \psi_t$ are conjunctive components. Then $l(\varphi) = max\{length(\psi_i) \mid 1 \leq i \leq t\}$ and $l(\Theta, \varphi) = l(\varphi) \cdot size(\Theta) \cdot |\Theta|$.*

Lemma 5.24 *Fix a disjunctive query $\varphi = \psi_1 \vee \psi_2 \vee \cdots \vee \psi_m$ where $\psi_1, \cdots, \psi_t$ are conjunctive components. Let Θ be a finite set of sequential r.e.'s $\theta_1, \cdots, \theta_s$. For each $n \geq l(\Theta, \varphi)$, $\cup_{1 \leq k \leq s} D_{\theta_k,n} \models \varphi$ if, and only if, $\cup_{1 \leq k \leq s} D_{\theta_k, l(\Theta,\varphi)} \models \varphi$.*

Proof "$\cup_{1 \le k \le s} D_{\theta_k,n} \models \varphi$ if $\cup_{1 \le k \le s} D_{\theta_k,l(\Theta,\varphi)} \models \varphi$" is obvious. We prove the opposite direction. Assume $\mathcal{D}_{l(\Theta,\varphi)} \not\models \varphi$ for some model $\mathcal{D}_{l(\Theta,\varphi)}$. We show that some model $\mathcal{D}_n$ of $\cup_{1 \le k \le s} D_{\theta_k,n}$, which is a suitable extension of $\mathcal{D}_{l(\Theta,\varphi)}$, holds $\mathcal{D}_n \not\models \varphi$.

Let us focus on a starred expression σ_i in $\sigma = \sigma_1\sigma_2\cdots\sigma_l \in \Theta$. Since the number of constant and starred expressions appearing in Θ is at most $size(\Theta)\cdot|\Theta|$, $\psi(\sigma_i)^{l(\Theta,\varphi)}$ has a segment longer than $l(\varphi)$ that overlaps with at most one starred expression. Let $\mathcal{D}'_{l(\Theta,\varphi)}$ be an extension of $\mathcal{D}_{l(\Theta,\varphi)}$ by inserting $\psi(\sigma_i)^{n-l(\Theta,\varphi)}$ to that segment. Then, $\mathcal{D}'_{l(\Theta,\varphi)} \not\models \varphi$ from $\mathcal{D}_{l(\Theta,\varphi)} \not\models \varphi$.

Repeating this process to each starred expression appearing in Θ, we obtain a model $\mathcal{D}_n$ of $\cup_{1 \le k \le s} D_{\theta_k,n}$, keeping $\mathcal{D}_n \not\models \varphi$.

The next theorem is immediate from Lemma 5.24.

Theorem 5.25 *Let Φ be a finite set of finite sets of sequential r.e.'s. Then,*

$$\texttt{ExistsMinor}(\Phi) = \vee_{\Theta\in\Phi}\ \texttt{QueryTest}(\cup_{\theta\in\Theta} D_{\theta,l(\Theta,\varphi)}).$$

Theorem 5.26 *The algorithm (in Fig. 5.2) to detect a set of minors $\mathcal{M}_\varphi$ terminates.*

Proof From Theorem 5.12, for each iteration of `while ExistsMinor(L)`, `L` strictly decreases wrt $\le_m$, and $<_m$ is an WFO from Corollary 5.13.

Thus, we can effectively compute a set of minors $\mathcal{M}_\varphi$, and obtain a simple algorithm to decide $D \models \varphi$.

Corollary 5.27 *For a fixed disjunctive monadic query φ, the linear time algorithm to decide whether $D \models \varphi$ for an indefinite database D is as follows:*

```
begin
  Flag:= false;
  for each $m$ in $\mathcal{M}_\varphi$ do Flag:= Flag or [$m \le_m D$];
  return flag;
end
```

Since the comparison wrt $\le_m$ can be done in linear-time to the size of D, this Corollary shows the generation of linear-time algorithm of fixed disjunctive query processing over indefinite databases.

5.3 WQO and regularity

In [EHR83], Ehrenfeucht et al. showed that a set L of finite words is regular if and only if L is $\le$-closed under some monotone well-quasi-order (WQO) $\le$ over finite words. We extend this result to regular ω-languages. That is,

1. an ω-language L is regular if and only if L is $\preceq$-closed under a *periodic* extension $\preceq$ of some monotone WQO over finite words, and

2. an ω-language L is regular if and only if L is $\preceq$-closed under a WQO $\preceq$ over ω-words that is a *continuous* extension of some monotone WQO over finite words.

Throughout this section, we will use A for a finite alphabet, A^* for a set of all (possibly empty) finite words on A, and A^ω for a set of all ω-words on A. A concatenation of two words u, v is denoted as u, v, an element-wise concatenation of two sets U, V of words by $U.V$, $\underbrace{V.V.\cdots.V}_{i}$ by V^i, and $V.V.V.\cdots$ by V^ω. The length of a finite word u is denoted by $|u|$. As a convention, we will use ϵ for the empty word, $u, v, w, \cdots$ for finite words, $\alpha, \beta, \cdots$ for ω-words, $a_1, a_2, \cdots$ for elements in A, $i, j, k, l, \cdots$ for indices, and $U, V, \cdots$ (capital letters) for sets. We sometimes use $x, y, \cdots$ for elements of a set.

5.3.1 ω-words

A regular ω-language is a set of ω-words that are accepted by a (nondeterministic) *Büchi* automaton $\mathcal{A} = \{Q, q_0, \Delta, F\}$, where Q is a finite set of states, q_0 an initial state, $\Delta \subseteq Q \times A \times Q$ a transition relation, and F a set of final states. $\alpha = a_1 a_2 a_3 \cdots \in A^\omega$ is accepted by $\mathcal{A}$ if its corresponding run $q_0 \underset{a_1}{\rightarrow} q_1 \underset{a_2}{\rightarrow} q_2 \underset{a_3}{\rightarrow} \cdots$ runs through some state of F infinitely often. A set of ω-words accepted by $\mathcal{A}$ is denoted as $L(\mathcal{A})$. For states q, q' and $w \in A^*$, we write $q \underset{w}{\rightarrow} q'$ if there is a run of $\mathcal{A}$ on w, and we write $q \overset{F}{\underset{w}{\rightarrow}} q'$ if there is a run of $\mathcal{A}$ on w from q to q' such that the run runs through some state of F.

A congruence $\sim$ is an equivalent relation over A^* preserved by concatenations. It is finite if there are only finitely many $\sim$-classes. Details are given elsewhere [Tho90b].

Definition 5.28 Let $L \subseteq A^\omega$ and let $\sim$ be a congruence over A^*. We say that $\sim$ saturates L if for each $\sim$-class U, V, $U.V^\omega \cap L \neq \phi$ implies $U.V^\omega \subseteq L$.

Lemma 5.29 For a *Büchi* automaton $\mathcal{A}$ and $u, v \in A^*$, we define $u \sim_{\mathcal{A}} v$ if $(q \underset{u}{\rightarrow} q' \Leftrightarrow q \underset{v}{\rightarrow} q') \wedge (q \overset{F}{\underset{u}{\rightarrow}} q' \Leftrightarrow q \overset{F}{\underset{v}{\rightarrow}} q')$ for each $q, q' \in Q$. Then $\sim_{\mathcal{A}}$ is a finite congruence that saturates $L(\mathcal{A})$.

Theorem 5.30 $L \subseteq A^\omega$ is regular if and only if some finite congruence saturates L.

Lemma 5.31 Let $\sim$ be a finite congruence over A^*.

1. Let $\alpha = u_1 u_2 \cdots \in A^\omega$ and let $u(i, j) = u_i u_{i+1} \cdots u_{j-1}$ where $u_i \in A^*$. There exist a $\sim$-class V and $i_1 < i_2 < \cdots$ such that $u(i_j, i_k) \in V$ for each j, k with $j < k$.

2. Let U, V be $\sim$-classes. There exist $\sim$-classes U', V' such that $U.V^\omega \subseteq U'.V'^\omega$, $U'.V' \subseteq U'$, and $V'.V' \subseteq V'$.

Proof

1. Since $\sim$ has only finitely many $\sim$-classes, this is a direct consequence of (infinite) *Ramsey Theorem.*

2. Note that for each $\sim$-class $U_1, \cdots, U_m, W$, $U_1. \cdots .U_n \cap W \neq \emptyset$ implies $U_1. \cdots .U_n \subseteq W$. Since $\sim$ has only finitely many $\sim$-classes, from (infinite) *Ramsey Theorem* there exist a $\sim$-class V' and $i_1 < i_2 < \cdots$ such that $V^{i_k - i_j} \subseteq V'$ for each j, k with $j < k$ and $V'.V' \subseteq V'$. Let U' be a $\sim$-class that includes $U.V^{i_1}$. Then $U.V^\omega \subseteq U'.V'^\omega$, $U'.V' \subseteq U'$, and $V'.V' \subseteq V'$. ■

For a QO $(S, \leq)$ and $L \subseteq S$, L is $\leq$-closed if $x \in L$ and $x \leq y$ imply $y \in L$. A QO $(A^*, \leq)$ is monotone if $u \leq v$ implies $w_1 u w_2 \leq w_1 v w_2$ for each $u, v, w_1, w_2 \in A^*$. As a convention, a QO over finite words is denoted as $\leq$, and a QO over ω-words is denoted as $\preceq$.

5.3.2 First theorem

Definition 5.32 A QO $(A^\omega, \preceq)$ is a periodic extension of $(A^*, \leq)$ if the following conditions are satisfied:

- For each $u_i, v_i \in A^*$, $u_i \leq v_i$ for any i imply $u_1 u_2 u_3 \cdots \preceq v_1 v_2 v_3 \cdots$.
- For each $\alpha \in A^\omega$, there exist $u, v \in A^*$ such that $\alpha \preceq u.v^\omega$ and $\alpha \succeq u.v^\omega$.

Theorem 5.33 Let $L \subseteq A^\omega$. L is regular if and only if L is $\preceq$-closed under a periodic extension $(A^\omega, \preceq)$ of a monotone WQO $(A^*, \leq)$.

For instance, the embedding over ω-words is the periodic extension of the embedding over finite words. Note that a periodic extension of a monotone WQO over A^* is a WQO over A^ω. We will prove Theorem 5.33 below.

Lemma 5.34 Let $\sim$ be a finite congruence on A^* and let U, V be $\sim$-classes. For $u, v \in A^*$, if $uv^\omega \in U.V^\omega$, $U.V \subseteq U$, and $V.V \subseteq V$, there exist $w_1 \in U$ and $w_2 \in V$ such that $w_1 w_2^\omega = uv^\omega$.

Proof Let $uv^\omega = u'v'_1v'_2 \cdots$ satisfying $u' \in U$ and $v'_i \in V$, and let $w(i,j) = v'_i \cdots v'_{j-1}$ for $i < j$. Let $k_j \equiv |w(1,j)|$ $(mod\ |v|)$. Then there exist k_{j_1} and k_{j_2} such that $k_{j_1} < k_{j_2}$ and $k_{j_1} \equiv k_{j_2}$ $(mod\ |v|)$. Since there are infinitely many such pairs, we can assume that $|u| \leq |u'w(1, j_1 - 1)|$. Let $w_1 = u'.w(1, j_1 - 1)$ and $w_2 = w(j_1, j_2 - 1)$. Since $U.V \subseteq U$ and $V.V \subseteq V$, $w_1 \in U$, $w_2 \in V$ and $uv^\omega = w_1 w_2^\omega$. ■

Lemma 5.35 For a *Büchi* automaton $\mathcal{A}$ and $\alpha \in A^\omega$, let $[\![\alpha]\!] = \{U.V^\omega \mid \alpha \in U.V^\omega\}$ where U, V are $\sim_\mathcal{A}$-classes. We define $\alpha \preceq' \beta$ if $[\![\alpha]\!] \cap [\![\beta]\!] \neq \emptyset$. Then,

1. $L(\mathcal{A})$ is $\preceq'$-closed.
2. $u_i \sim_\mathcal{A} v_i$ for each i imply $u_1 u_2 \cdots \preceq' v_1 v_2 \cdots$.

Proof From Lemma 5.29, $\sim_\mathcal{A}$ saturates L and $U.V^\omega \subseteq L$ for each $U.V^\omega \in [\![\alpha]\!]$. Thus L is $\preceq'$-closed.

From Lemma 5.31 (i), there exist a $\sim_\mathcal{A}$-class V and $i_1 < i_2 < \cdots$ such that $u(i_j, i_k) \in V$ for each $j < k$. Let U be a $\sim_\mathcal{A}$-class such that $u(1, i_1) \in U$. (We borrow the notation from Lemma 5.31 (i).) Since $\sim_\mathcal{A}$ is a congruence, $v(1, i_1) \in U$ and $v(i_j, i_k) \in V$ for each $j < k$. Thus $u_1 u_2 \cdots \in U.V^\omega$ implies $v_1 v_2 \cdots \in U.V^\omega$, and $\alpha \preceq \beta$. ■

Definition 5.36 [Arn85] For $u, v \in A^*$, we define $u \approx_L v$ if $w(w_1uw_2)^\omega \in L \Leftrightarrow w(w_1vw_2)^\omega \in L$ and $w_1uw_2w^\omega \in L \Leftrightarrow w_1vw_2w^\omega \in L$ for each $w, w_1, w_2 \in A^*$.

Proof of Theorem 5.33
Only-if part: Assume L is regular. Let $\mathcal{A}$ be a *Büchi* automaton such that $L = L(\mathcal{A})$. Since $\sim_\mathcal{A}$ is a finite congruence, $(A^*, \sim_\mathcal{A})$ is a monotone WQO. Define $\preceq$ as the transitive closure of $\preceq'$ (defined in Lemma 5.35), then $(A^\omega, \preceq)$ is a periodic extension of $(A^*, \sim_\mathcal{A})$ and $L(\mathcal{A})$ is $\preceq$-closed.

If part: Assume that L is $\preceq$-closed where $\preceq$ is a periodic extension of a monotone WQO $\leq$. First, we show that $\approx_L$ is a finite congruence. Assume that $\{u_i\}$ is an infinite set in A^* such that $u_i \not\approx_L u_j$ for $i \neq j$. Since $(A^*, \leq)$ is a WQO, there exists an infinite ascending subsequence $\{u_{k_i}\}$.

Let $F(u) = \{(v, v_1, v_2, w_1, w_2, w) \in A^* \times A^* \times A^* \times A^* \times A^* \times A^* \mid v(v_1uv_2)^\omega \in L \wedge w_1uw_2w^\omega \in L\}$. Since $\preceq$ is a periodic extension of $\leq$ and L is $\preceq$-closed, each $F(u)$ is $\leq \times \leq \times \leq \times \leq \times \leq \times \leq$-closed and hence $F(u_{k_i}) \subseteq F(u_{k_j})$ for $i < j$. Since $u_{k_i} \not\approx_L u_{k_j}$ for $i \neq j$, $F(u_{k_i}) \neq F(u_{k_j})$, thus $F(u_{k_i}) \subset F(u_{k_j})$. Then there exists an infinite sequence in which each pair of different elements is incomparable. Since $\leq \times \leq \times \leq \times \leq \times \leq \times \leq$ is a WQO over $A^* \times A^* \times A^* \times A^* \times A^* \times A^*$, this is a contradiction.

Second, we show that $\approx_L$ saturates L. Assume that some $\approx_L$-classes U, V satisfy $U.V^\omega \cap L \neq \phi$ and $U.V^\omega \not\subseteq L$. From Lemma 5.31 (ii), we can assume that $U.V \subseteq U$ and $V.V \subseteq V$.

Let $\alpha \in U.V^\omega \cap L$ and $\beta \in U.V^\omega \setminus L$. Since $(A^\omega, \preceq)$ is a periodic extension, from Lemma 5.34 there exist $u, u' \in U$ and $v, v' \in V$ such that $\alpha = uv^\omega$ and $\beta = u'v'^\omega$. By definition of $\approx_L$, $uv^\omega \in L$ and $u'v'^\omega \notin L$ are contradictory. ■

5.3.3 Second theorem

Definition 5.37 For a monotone QO $(A^*, \leq)$, a QO $(A^\omega, \preceq)$ is a *continuous extension* if the following conditions are satisfied.

1. For each $u, v \in A^*$ and $\alpha, \beta \in A^\omega$, $u \leq v$ and $\alpha \preceq \beta$ imply $u\alpha \preceq v\beta$.

2. Let $u_j, v_j \in A^*$ for each j and let $\alpha_i = v_1 \cdots v_{i-1}u_i \cdots$ for each i and $\alpha_\infty = v_1v_2 \cdots$. For $\beta \in A^\omega$, if $u_i \leq v_i$ and $\alpha_i \preceq \beta$ for each i, then $\alpha_\infty \preceq \beta$, and if $u_i \geq v_i$ and $\alpha_i \succeq \beta$ for each i, then $\alpha_\infty \succeq \beta$.

Theorem 5.38 Let $L \subseteq A^\omega$. L is regular if and only if L is $\preceq$-closed under a WQO $(A^\omega, \preceq)$ that is a continuous extension of a monotone WQO $(A^*, \leq)$.

For the embedding $\leq$ over finite words, let $(A^*, \leq^\circ)$ be defined as $u \leq^\circ v$ if and only if $u \leq v$ and $elt(u) = elt(v)$, where $elt(u) = \{a_i \mid u = a_1a_2 \cdots a_j\}$. Since the embedding $\leq$ over finite words is a WQO from Higman's lemma, $\leq^\circ$ is also a WQO. Then the embedding over A^ω is a continuous extension of $\leq^\circ$. Note that the embedding over A^ω is a continuous extension of the embedding $\leq$ over finite words. Actually, any continuous extension of the embedding $\leq$ over finite words is a trivial WQO (i.e., $A^\omega \times A^\omega$). For instance, given $\alpha, \beta \in A^\omega$. Let $\alpha(1,i)$ be the prefix of α of the length i and $\alpha_i = \alpha(1,i).\beta$ for each i. Since $\alpha(1,i) \geq \epsilon$, $\alpha_i \succeq \beta$ for each i. Thus, by definition of continuity, $\alpha_\infty = \alpha \succeq \beta$. Hence, for any $\alpha, \beta \in A^\omega$, we conclude $\alpha \succeq \beta$.

Definition 5.39 Let $u, v \in A^*$ and let $L \subseteq A^\omega$. We write

- $u \simeq_L^1 v$ if and only if $\forall w \in A^*, \forall \alpha \in A^\omega.\ wu\alpha \in L \Leftrightarrow wv\alpha \in L$,
- $u \simeq_L^2 v$ if and only if $\forall w \in A^*\ .\ wu^\omega \in L \Leftrightarrow wv^\omega \in L$, and
- $u \simeq_L v$ if and only if $u \simeq_L^1 v$ and $u \simeq_L^2 v$.

Proof of Theorem 5.38
Only-if part: Assume L is regular. Let $\mathcal{A}$ be a *Büchi* automaton such that $L = L(\mathcal{A})$. Since $\sim_\mathcal{A}$ is a finite congruence, $(A^*, \sim_\mathcal{A})$ is a monotone WQO. Define $\preceq$ as the transitive closure of $\preceq'$ (defined in Lemma 5.35), then $L(\mathcal{A})$ is $\preceq$-closed. Since $\preceq'$ is symmetric, $(A^\omega, \preceq)$ is a continuous extension of $(A^*, \sim_\mathcal{A})$ from Lemma 5.35 (ii). For the index n of $\sim_\mathcal{A}$, the number of $\preceq$-classes is bound by 2^{n^2}. Thus $\preceq$ is a WQO.

If part: First, we show that $\simeq_L$ is a finite congruence. Assume that $\{u_i\}$ is an infinite set in A^* such that $u_i \not\simeq_L u_j$ for $i \neq j$. Since $(A^*, \leq)$ is a WQO, there exists an infinite ascending subsequence $\{u_{k_i}\}$.

Let $F(u) \subseteq A^* \times A^\omega \times A^*$ be a set such that $(w, \alpha, v) \in F(u) \Leftrightarrow wu\alpha \in L \wedge vu^\omega \in L$. Then, each $F(u)$ is $\leq \times \preceq \times \leq$-closed and hence $F(u_{k_i}) \subseteq F(u_{k_j})$ for $i < j$. Since $u_{k_i} \not\simeq_L u_{k_j}$ for $i \neq j$, $F(u_{k_i}) \neq F(u_{k_j})$, thus $F(u_{k_i}) \subset F(u_{k_j})$. Then there exists an infinite sequence in which each pair of different elements is incomparable. Since $\leq \times \preceq \times \leq$ is a WQO over $A^* \times A^\omega \times A^*$, this is a contradiction.

Second, we show that $\simeq_L$ saturates L. Assume that some $\simeq_L$-classes U, V satisfy $U.V^\omega \cap L \neq \phi$ and $U.V^\omega \not\subseteq L$. From Lemma 5.31 (2), we can assume that $V.V \subseteq V$.

Let $\alpha = uv_1v_2\cdots$ be a minimal element (wrt $\preceq$) in $U.V^\omega \cap L$, and let $\beta = u'v'_1v'_2\cdots \in U.V^\omega \setminus L$ such that $u, u' \in U$ and $v_i, v'_i \in V$. Let $\{\bar{v}_l\}$ be sets of minimal elements of V wrt $\leq$. Since $(V, \leq)$ is a WQO, $\{\bar{v}_l\}$ are finite.

Let $\alpha'(j, j+k) = v_j \cdots v_{j+k}$. Since $\bar{v}_l$ are finitely many, from (infinite) *Ramsey Theorem* there exist l and an ascending sequence $0 < j_1 < j_2 < \cdots$ such that $\alpha'(j_m, j_{m+1} - 1) \geq \bar{v}_l$ for any $m > 0$.

Let $\alpha_m = u\ \alpha'(1, j_1 - 1)\ \bar{v}_l^{m-1}\ \alpha'(j_m, j_{m+1} - 1)\cdots$. Obviously, $\alpha_m \preceq \alpha$ and $\alpha_m \in U.V^\omega \cap L$. Since α is minimal in $U.V^\omega \cap L$, $\alpha_m \succeq \alpha$. By definition of the continuous extension, $\alpha_\infty = u\ \alpha'(1, j_1 - 1)\ \bar{v}_l^\omega \succeq \alpha$. Thus, since L is $\preceq$-closed, $\alpha_\infty \in U.V^\omega \cap L$.

Let $\beta'(j, j+k) = v'_j \cdots v'_{j+k}$. Since $\bar{v}_l$ are finitely many, from (infinite) *Ramsey Theorem* there exist l' and an ascending sequence $0 < j'_1 < j'_2 < \cdots$ such that $\beta'(j'_m, j'_{m+1} - 1) \geq \bar{v}_{l'}$ for any $m > 0$. Let $\beta_\infty = u'\ \beta'(1, j_1 - 1)\ \bar{v}_{l'}^\omega$. By definition of the continuous extension, $\beta_\infty \preceq \beta$. Since L is $\preceq$-closed, $\beta \notin L$ implies $\beta_\infty \notin L$. Thus $\bar{\beta} \in U.V^\omega \setminus L$.

Since $u \simeq_L^1 u'$ and $\bar{v}_j \simeq_L^2 \bar{v}_{j'}$ for each j, repeated applications of $\simeq_L^1$ and an application of $\simeq_L^2$ imply that $\alpha_\infty \in L$ if and only if $\beta_\infty \in L$. This contradicts $\alpha_\infty \in L$ and $\beta_\infty \notin L$.
■

Example 5.40 Either the periodic or continuous assumption in the theorems is needed to conclude regularity. Let $\beta = abaabaaabaaaab\cdots$ and let $L(\beta)$ be the set of ω-words that have a common suffix with β. For $\alpha \in A^\omega$, let $p_\beta(\alpha) = 1$ if $\alpha \in L(\beta)$ and let $p_\beta(\alpha) = 0$ if $\alpha \notin L(\beta)$. Define $\alpha \preceq \alpha' \Leftrightarrow p_\beta(\alpha) \leq p_\beta(\alpha')$. Then $\preceq$ is a WQO over ω-words and $L(\beta)$ is $\preceq$-closed, but $L(\beta)$ is not regular.

5.4 Possible improvement by fold/unfold program transformation

This chapter described the generation of a linear time query-processing algorithm for a fixed disjunctive monadic query in an indefinite database on a linearly ordered domain. This problem was first posed by Van der Meyden [van97] and had, until now, not been reported elsewhere. There are several future directions:

1. Our method is based on the regular expression techniques in Murthy-Russell's constructive proof of Higman's lemma [MR90]. Among its known constructive proofs [MR90, RS93, CF93] (or intuitionistic proofs [Ges96, Fri96]), [CF93] would be one of the most simple and is implemented on Coq prover[4]. This could be applied to a simpler method of algorithm generation, in combination with well-developed proof-extraction techniques.

2. We are designing an automatic generator based on MONA.[5] MONA, which runs on Linux, efficiently decides the satisfiability of formulae in monadic second order logic S1S/S2S. For efficient implementation, we also need some reduction of sequential r.e.'s to smaller and equivalent sequential r.e.'s by removing redundancies.

3. The next extension may be to use the constructive proof of Kruskal's theorem [NW63] for more general problems. Gupta demonstrated the constructive proof of the weaker form [Gup92], and recently Veldman presented an intuitionistic proof of Kruskal's theorem [Vel00]. These would correspond to, for instance, query-processing in an indefinite database over partial ordered domains (i.e., events on branching time).

Adding to topics listed above, we may need to reduce the constant of a generated linear time algorithm to make it practical. For instance, as the example of the disjunctive query $\varphi = \psi_1 \vee \psi_2 \vee \psi_3$ (Section 5.2) where

$$\begin{cases} \psi_1 &= \exists xyz[P(x) \wedge Q(y) \wedge R(z) \wedge \ x < y < z], \\ \psi_2 &= \exists xyz[Q(x) \wedge R(y) \wedge P(z) \wedge \ x < y < z], \text{ and} \\ \psi_3 &= \exists xyz[R(x) \wedge P(y) \wedge Q(z) \wedge \ x < y < z]. \end{cases}$$

[4]See `http://coq.inria.fr/contribs/logic-eng.html`.
[5]Available at `http://www.brics.dk/mona`.

suggests the number of minors would easily explode. The example requires comparison with 10 minors

$$\begin{array}{l}
\{\ \{P\}\{Q\},\ \{Q\}\{R\},\ \{R\}\{P\}\ \},\ \{\ \{P\}\{Q\}\{R\}\ \},\ \{\ \{Q\}\{R\}\{P\}\ \},\ \{\ \{R\}\{P\}\{Q\}\ \},\\
\{\ \{P\}\{Q\}\{P\},\ \{Q\}\{R\}\ \},\ \{\ \{Q\}\{R\}\{Q\},\ \{R\}\{P\}\ \},\ \{\ \{R\}\{P\}\{R\},\ \{P\}\{Q\}\ \}.\\
\{\ \{P\}\{R\}\{P\},\ \{Q\}\{R\}\ \},\ \{\ \{Q\}\{P\}\{Q\},\ \{R\}\{P\}\ \},\ \{\ \{R\}\{Q\}\{R\},\ \{P\}\{Q\}\ \}.
\end{array}$$

We expect that a program transformational approach, such as stratification by folding, would be useful in reducing the number of comparisons with minors. For instance, the number of the tests will be reduced (from 10 times to at most 6) if the comparison is stratified as follows.

$$\begin{tikzcd}
\textit{query} \arrow[d, "?"] & \\
\{\ \{P\}\{Q\},\ \{Q\}\{R\}\ \} \arrow[r, "yes"] \arrow[d, "No"] & \left\{\begin{array}{l} \{\ \{P\}\{Q\},\ \{Q\}\{R\},\ \{R\}\{P\}\ \},\ \{\ \{P\}\{Q\}\{R\}\ \},\\ \{\ \{P\}\{Q\}\{P\},\ \{Q\}\{R\}\ \},\ \{\ \{R\}\{Q\}\{R\},\ \{P\}\{Q\}\ \}.\end{array}\right. \\
\{\ \{Q\}\{R\},\ \{R\}\{P\}\ \} \arrow[r, "yes"] \arrow[d, "No"] & \left\{\begin{array}{l} \{\ \{Q\}\{R\}\{P\}\ \},\\ \{\ \{Q\}\{R\}\{Q\},\ \{R\}\{P\}\ \},\ \{\ \{P\}\{R\}\{P\},\ \{Q\}\{R\}\ \}.\end{array}\right. \\
\{\ \{R\}\{P\},\ \{P\}\{Q\}\ \} \arrow[r, "yes"] & \left\{\begin{array}{l} \{\ \{R\}\{P\}\{Q\}\ \},\\ \{\ \{R\}\{P\}\{R\},\ \{P\}\{Q\}\ \},\ \{\ \{Q\}\{P\}\{Q\},\ \{R\}\{P\}\ \}.\end{array}\right.
\end{tikzcd}$$

Chapter 6

Automatic generation based on tree decomposition

A graph is a flexible relational structure for describing problems, but solving such problems is often not so easy. Difficulty arises because an efficient implementation of a graph is not obvious. In mathematics, a graph is expressed by a pair of a vertex set and an edge set. This is clear, but a set as a data structure is a source of inefficiency from the algorithmic view point. Some algorithms (especially those for network flow problems) express a graph by a matrix, but even for a sparse graph the size of the matrix grows to the square to the size of a graph.

Recently, graphs with bounded tree width [RS86] (or, partial k-tree [ACPS93], the same concept appeared simultaneously) have been receiving attention because of their interesting algorithmic properties, such as the fixed parameter tractability [DF95, Flu01, Gro01]. That is, the complexity of inherently difficult graph problems is often estimated as $O(2^{twd(G)^{\alpha}} \cdot |V(G)|)$ for a graph G, its tree width $twd(G)$, and its vertex set $V(G)$. This suggests that the restriction of graphs with the fixed upper bound of tree width (called *bounded tree width*) can make NP-hard graph algorithms (such as Hamiltonian circuits and chromatic numbers) be even linear-time solvable. Actually, we know that linear time solutions for monadic second order definable properties (such as reachability and connectivity, see the list of properties in [Cou90]) are automatically generated for graphs with bounded tree width [BLW87, Cou90, BPT92, SHTO00], as illustrated in Fig. 6.1.

This is excellent in theory, but still infeasible in practice, because of the huge constants of generated linear time algorithms. Such constants easily explode to the tower of exponentials. Our basic approach is specification by *finite mutumorphism*, instead of that by formulae. Then, fusion/tuppling program transformation of functional programs would drastically reduce the constant.

Section 6.1 shows simple but non-trivial examples of our methods [SHTO00, SHTO02]; *maximum segment sum problem* (MSS) and *k-maximum segment sum problem* (k-MSS). The generated linear time programs are as fast as hand-coded ones [Ben84, BRS99], both in theory and practice.

Their back born structures remain in simple graphs, such as words and trees. To extend the scope as in Section 6.4, we face the problem that the algebraic construction of

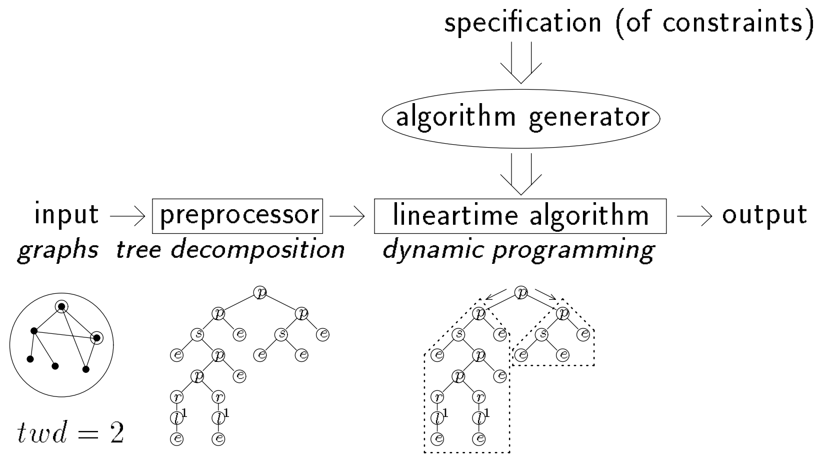


Figure 6.1: Linear time algorithm generation for graphs with bounded tree width

graphs (such as tree decomposition) is *not* initial, whereas most of program calculational techniques assume an initial algebra. This problem was already found in the derivation of the Dijkstra algorithm on the shortest path problem [篠00]. Thus, we need the extensions of fusion/tuppling program transformational techniques to non-initial algebra, which were suggested in [Fok96].

For this purpose, Section 6.2 presents the complete axiomatization of the algebraic construction of graphs [ACPS93]. At the moment, whether some finite fragment of this infinite axiomatization is complete for graphs with bounded tree-width is not clear; but I strongly believe such completeness holds and this will be the next step.

6.1 Examples of linear time algorithms for data mining

Data mining, which is a technology for obtaining useful knowledge from large database, has been gradually recognized as an important subject. There have been developed many efficient algorithms for various kinds of data mining problems; here we concentrate on the problem of mining optimized association rules [BRS99, FMMT96b, FMMT96a].

The core of mining optimized gain association rules problem is transformed to the problem called *maximum segment sum problem* (MSS for short) [FMMT96b], and we have a linear time algorithm [Ben84]. Input of the MSS problem is a number list xs, and output is a segment (consecutive sublist) of xs that has the maximum sum among all segments of xs. For example, in the case of $xs = [5, -10, 20, 5, -15, 30, -5]$, the result is $[20, 5, -15, 30]$, which has the maximum sum 40.

6.1.1 MSS problem

The hand-coded linear time program by Bentley is shown in Fig. 6.2 (which is originally a C program and reorganized as a Haskell program) [Ben84]. In Fig. 6.2, `mssloop` scans

```
mss xs = mssloop xs 0 0

mssloop []     mval val = max mval val
mssloop (x:xs) mval val = let newval = max 0 (val + x)
                           in mssloop xs (max mval newval) newval
```

Figure 6.2: Program of MSS by Bentley

the input list `xs` from the head to the end. `mval` expresses the maximum sum in the currently scanned prefix (which is expressed as `MaxSoFar` in Algorithm 4 in [Ben84]), and `val` expresses the maximum value among the sums of suffixes in the currently scanned prefix (which is expressed as `MaxEndingHere`).

As an (easy) instance of the automatic generation based on tree decomposition [BLW87, Cou90, BPT92], the linear time algorithm of MSS is automatically generated; however its cost is still too expensive. That is, the constant grows up to $2^{2^{2^{2^{2^8}}}}$. Our aim is to apply program transformational techniques, such as fusion and tuppling transformation [Fok89, MFP91, TM95, HITT97], to reduce the constant. For MSS, we have successfully reduced the constant to 2^3 [SHTO00], which is comparable to the hand coded program above [Ben84, Gri90].[1]

Let marked elements express a (possibly non-contiguous) sublist in a list. A marked element `x` is represented as `(x,True)`, and `(x,False)` otherwise. Starting from the *finite mutumorphism* specification of connectivity `conn`

```
conn [x] = True
conn (x:xs) = if marked x then nm xs || (marked (head xs) && conn xs)
                          else conn xs

nm [x] = not (marked x)
nm (x:xs) = not (marked x) && nm xs
```

we obtain the classification consists of all possible combinations of the results of 3 boolean-valued functions that characterize connectivity. More specifically, these functions are: `conn`, marked elements in the list are contiguous, `nm`, no elements in the list are marked, and `mh`, the first element in the list are marked. Their values correspond to `(c,n,m)` (in this order) in Fig. 6.3. The number of their possible combination is 2^3 and for each combination the maximal candidate is computed. After scanning the input list, we can choose the maximum value from combinations satisfying `conn`, which is filtered by the accepting function `accept_mss`.

Fig. 6.3 presents the generated Haskell program. `opt` is the generic function for maximum weight sum problems, presented in Fig. 6.4. Refer [SHTO00], for details.

[1]Similar for the party planning problem [CLR90, BdM96].

```
mss = opt accept_mss phi1_mss phi2_mss

accept_mss (c,n,m) = c

phi1_mss mx = (True, not (marked mx), marked mx)

phi2_mss mx (c,n,m) = (if marked mx then (n || (m && c)) else c,
                       not (marked mx) && n,
                       marked mx)
```

Figure 6.3: Automatically generated program of MSS

```
opt accept phi1 phi2 xs =
    let opts = candidates phi1 phi2 xs
     in third (getmax [(c,w,t) | (c,w,t) <- opts, accept c])

candidates phi1 phi2 [x] =
        eachmax [(phi1 mx, if marked mx then weight mx else 0, [mx])
                   | mx <- [mark x, unmark x]]

candidates phi1 phi2 (x:xs) =
    let opts = candidates phi1 phi2 xs
     in eachmax [(phi2 mx c, (if marked mx then weight mx else 0) + w,
                  mx:cand)
                  | mx <- [mark x, unmark x], (c,w,cand) <- opts]

getmax [] = error "No solution."
getmax xs = foldr1 f xs
            where f (c1,w1,cand1) (c2,w2,cand2) =
                         if w1>w2 then (c1,w1,cand1) else (c2,w2,cand2)

eachmax xs = foldl f [] xs
             where f [] (c,w,cand) = [(c,w,cand)]
                   f ((c',w',cand') : opts) (c,w,cand) =
                         if c==c' then if w>=w' then (c,w,cand) : opts
                                                else (c',w',cand') : opts
                                  else (c',w',cand') : f opts (c,w,cand)
```

Figure 6.4: Definition of the generic function `opt` for the maximum weight sum problem

6.1.2 k-MSS problem

The extension of MSS, called k-MSS, is used to solve the extension of the mining optimized association rules [BRS99]. Input of the k-MSS problem is a number list xs, and output is the (at most) k-segments of xs that have the maximum sum among all other choices from xs. For example, in the case of $xs = [5, -10, 20, 5, -15, 30, -5]$ and k=2, the result consists of $[20, 5]$ and $[30]$, which have the maximum sum 55. [BRS99] proposed the linear time algorithm (both linear to k and the input size) of k-MSS. The algorithm is a k-path algorithm, and at each i-th path a solution of i-MSS is obtained.

- $i = 1$: At the first path, solve 1-MSS as in [Ben84].
- $i > 1$: Let the solution of the $(i-1)$-MSS be $s_1, s_2, \ldots, s_{i-1}$ and the remaining sublists be $t_1, t_2, \ldots, t_j$ with $i - 1 \leq j \leq i + 1$. Solve 1-MSS for $t_1, t_2, \ldots, t_j$ and let one that has the maximum solution be t_{max}. Solve 1-minimum segment sum problem for $s_1, s_2, \ldots, s_{i-1}$ and let one that has the minimum solution be s_{min}. If the segment sum of t_{max} plus the segment sum of s_{min} is less than 0, then split s_{min} into three subintervals with the solution of s_{min} as the middle interval and delete s_{min} from the solution of $(i-1)$-MSS and add the first and third intervals to it, which gives the solution of i-MSS. Otherwise, split t_{max} into three subintervals with the solution of t_{max} as the middle interval and add the solution of t_{max} to the solution of $(i-1)$-MSS, which gives the solution of i-MSS.

This algorithm iterates k times the process of finding the most effective sublist and splitting it, and its complexity is $O(kn)$.

Our method in [SHTO00] generates the linear time algorithm; the property description that the marked elements form k-contiguous components is given below for $1 \leq i \leq k$. The intuition behind is; `p_i xs` returns `True` if the marked elements in `xs` compose at most i-contiguous components, and returns `False` otherwise.

```
p_i [x]    = True
p_i (x:xs) = if marked x
                then if marked (hd xs) then p_i xs
                                       else p_i-1 xs
                else p_i xs
p_0 [x]    = not (marked x)
p_0 (x:xs) = not (marked x) and p_0 xs
```

The generated program first computes the maximum value for each possible combination of the result of $k+2$ boolean valued functions, `p_i` for $0 \leq i \leq k$, `marked`, and `marked hd`, next filters the maximum values of the cases that fit to the property description, and choose the maximal value among them. Thus, the program is linear to the size of the input and the size of the possible combinations; 2^{k+3}. Thus, the complexity is $O(2^k n)$, which is much worse than that in [BRS99].

By supplementing the accumulators in the specification, we obtain the automatic generation of the linear time algorithm (both linear to k and the input size) of k-MSS [SHTO02]. This is done by extending the generic algorithm `opt` from Fig. 6.4

to Fig. 6.5 [SHT01] and the specification from mutumorphisms to functions of the form

$$p\ xs\ =\ g\ (foldr_h\ (\phi_1, \phi_2)\ \delta\ xs\ e_0)$$

where g is a function from *multi-marking* to boolean values and $foldr_h$ is the higher-order version of $foldr$ defined as

$$\begin{array}{lcl} foldr_h\ (\phi_1, \phi_2)\ \delta\ [\,]\ e & = & \phi_1\ e \\ foldr_h\ (\phi_1, \phi_2)\ \delta\ (x : xs)\ e & = & \phi_2\ x\ e\ (foldr_h\ (\phi_1, \phi_2)\ \delta\ xs\ (\delta\ x\ e)). \end{array}$$

Multi-marking is an extension of marking, that is, allowing an index of finite values instead of boolean values to each element. For k-MSS, an index of boolean values (here we consider `True` as `1` and `False` as `2`) is enough, but we essentially need the accumulator e in the definition of $foldr_h$ to reduce the number of function symbols (from k to 2) in the specification of k-MSS. For instance, the specification for k-MSS with an accumulator parameter is rewritten as below.

```
p xs = p' xs (2, k)

p' [] (m,e) = True
p' (x:xs) (m,e) = if m==1 then p' xs (markKind x, e)
                          else if markKind x
                               then if e > 0 then p' xs (1, e-1)
                                             else False
                               else p' xs (2, e)

markKind (x,m) = m
```

This specification is transformed to the program with $foldr_h$ by program calculational techniques, such as fusion and tuppling transformations. Actually, we successfully obtain the following specification by MAG system [dMS99], which is the system for automatic program transformation based on higher order pattern matching.

```
g x = x
phi1 (m,e) = True
phi2 x (m,e) r = if m==1 then r
                         else if markKind x
                              then if e > 0 then r else False
                              else r
delta (m,e) = if m==1 then if markKind x then (1,e) else (2,e)
                      else if markKind x then (1,e-1) else (2,e)
e_0 = 0
```

Then, $O(kn)$ algorithm of k-MSS is automatically generated as

```
opt 2 accept (f,+,0) phi1 phi2 delta
```

where

```
accept (c,e) = c && e == (2,k)
f x = if marked x then w x else 0
w (x,m) = x
```

```
opt k accept (f, oplus, id_oplus) phi1 phi2 delta xs =
  let opts = foldr psi2 psi1 xs
  in snd (getmax [(w,r) | Just (w,r) <- [ opts!i
                                        | i <- range bnds,
                                          opts!i /= Nothing,
                                          accept i]])
  where psi1 = array bnds [(i, g i) | i <- range bnds]
        psi2 x cand = accumArray h Nothing bnds
                        [((phi2 xm e c, e),
                          (f xm `oplus` w, xm:r))
                        | xm <- [(x,m) | m <- [1..k]],
                          e <- acclist,
                          ((c,_), Just (w,r)) <-
                             [ (i,cand!i)
                             | i <- [ (c',delta xm e)
                                    | c' <- classlist],
                               inRange bnds i,
                               cand!i /= Nothing]]
        g (c,e) = if (c == phi1 e) then Just (id_oplus, [])
                  else Nothing
        h (Just (w1,x1)) (w2,x2) = if w1 > w2 then Just (w1,x1)
                                   else Just (w2,x2)
        h Nothing (w,x) = Just (w,x)
        bnds = ((head classlist, head acclist)
                (last classlist, last acclist))

getmax [] = error "No solution."
getmax xs = foldr1 f xs
            where f (c1,w1,cand1) (c2,w2,cand2) =
                         if w1>w2 then (c1,w1,cand1) else (c2,w2,cand2)
```

Figure 6.5: Definition of the generic function `opt` for the maximum marking problem

6.2 Complete axiomatization for algebraic construction of graphs

We obtained success to generate a practical linear time algorithm for k-MSS. However, still the data structure of k-MSS is simple, i.e., sequence. These techniques are also applicable to trees, which compose an initial algebra. When we try to extend the method to more general graphs, we face to the problem that the algebra of graphs is *not* initial. Thus, the validity of the program calculational transformation rules is not clear yet. Our first step is to clarify the algebraic construction and the algebraic structure of graph.

Bauderon and Courcelle presented an algebraic construction of graphs [BC87, Cou90, CM01]. In general, a graph may have several algebraic constructions. Thus the algebraic structure of graphs should be clarified for the recursive computation to be well defined. [BC87, Cou90] also give the complete (infinite) axiomatization, but the size of their construction may grow square to the number of vertices; this is unwelcome situation from the algorithmic view point.

This section presents the complete (infinite) axiomatization for another algebraic construction of graphs. This algebraic construction is an extension of that in [ACPS93], of which the size grows linear to the number of vertices for graphs with bounded tree width [RS86]. For terminology for rewriting, refer Chapter 2.

6.2.1 Algebraic construction of graphs

A k-terminal graph G is a graph with a tuple of k vertices, called *terminals*. For simplicity, we consider simple graphs without loops (which are obtained as 0-terminal graphs after removal of terminals). The set of vertices of G is denoted by $V(G)$ and the set of edges of G is denoted by $E(G)$.

Definition 6.1 *k-terminal graphs G_1, G_2 are isomorphic if there exists a one-to-one onto map $\alpha : V(G_1) \to V(G_2)$ such that*

- *For $v \in V(G_1)$, if v is the i-th terminal of G_1 with $1 \leq i \leq k$, then $\alpha(v)$ is the i-th terminal of G_2, and vice versa.*
- *For $v, v' \in V(G_1)$, if (v, v') is an edge of G_1, then $(\alpha(v), \alpha(v'))$ is an edge of G_2, and vice versa.*

Definition 6.2 *Let B_k be sorts for $k \geq 0$. Let $l_k^i, p_k, r_k, \sigma_k^i, e^2, \mathbf{0}$ be signatures with sorts below*

$$\begin{cases} e^2 \colon B_2, & l_k^i : B_{k-1} \to B_k, \quad p_k \colon B_k \times B_k \to B_k, \\ \mathbf{0} : B_0, & r_k \colon B_k \to B_{k-1}, \quad \sigma_k^j \colon B_k \to B_k. \end{cases}$$

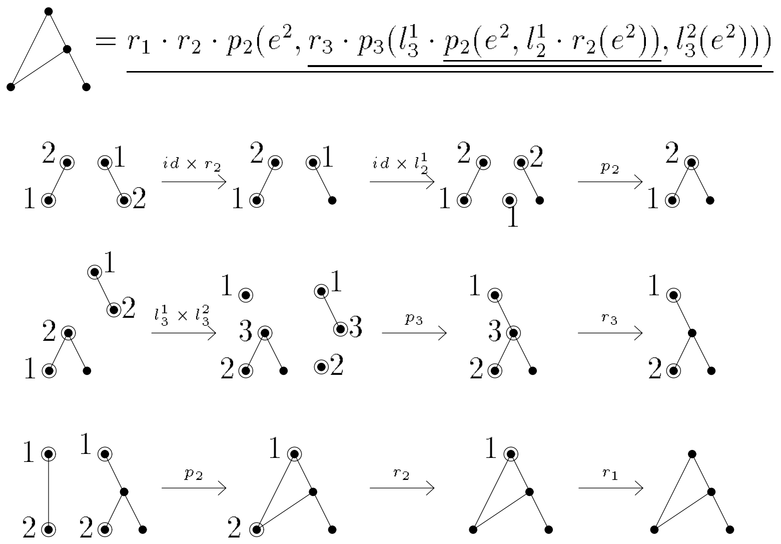
where $i \leq k$, $j < k$, and $k \geq 0$. Let $\mathcal{B}_n = T(\{\mathbf{0}, e^2, l_k^i, r_k, p_k, \sigma_k^i \mid 1 \leq i \leq k \leq n\})$.

A term $t \in \mathcal{B}_k$ is interpreted as a k-terminal graph (defined below) by interpreting signatures $l_k^i, p_k, r_k, \sigma_k^i, e^2, \mathbf{0}$ as following operations. This interpretation is denoted by $\psi(t)$. A term t is *connected* if $\psi(t)$ is a connected graph.

Definition 6.3 *Let e^2 be an edge with two terminals and $\mathbf{0}$ be an empty graph. We define operations among k-terminal graphs as*

- $\cdot$ $l_k^i(s)$ *is a lifting for* $1 \leq i \leq k$, *i.e., insert a new isolated terminal at the i-th position in* $k-1$ *terminals.*
- $\cdot$ $r_k(t)$ *removes the last-terminal.*
- $\cdot$ $p_k(s,t)$ *is a parallel composition for* $k \geq 0$, *i.e., fuse each i-th terminal in s and t for* $1 \leq i \leq k$.
- $\cdot$ $\sigma_k^i(t)$ *is a permutation, i.e., permute the i-th terminal and the* $i+1$*-th terminal in t for* $1 \leq i < k$.

Example 6.4 *Fig. 6.6 shows that the algebraic construction of a (0-terminal) graph. Each operation, underlined in* $r_1 \cdot r_2 \cdot p_2(e^2, r_3 \cdot p_3(l_3^1 \cdot p_2(e^2, l_2^1 \cdot r_2(e^2)), l_3^2(e^2)))$, *is figured in lower columns.*



$$= \underline{r_1 \cdot r_2 \cdot p_2(e^2, \underline{\underline{r_3 \cdot p_3(l_3^1 \cdot \underline{p_2(e^2, l_2^1 \cdot r_2(e^2))}, l_3^2(e^2))}})}$$

Figure 6.6: An example of the algebraic construction

Definition 6.5 *Two terms s, t with sort* B_k *are* equivalent *if the k-terminal graphs* $\psi(s), \psi(t)$ *are isomorphic.*

Example 6.6 $r_1 \cdot r_2 \cdot p_2(e^2, r_3 \cdot p_3(l_3^1 \cdot p_2(e^2, l_2^1 \cdot r_2(e^2)), l_3^2(e^2)))$ *and* $r_1 \cdot r_2 \cdot p_2(p_2(e^2, l_2^1 \cdot r_2(e^2)), r_3 \cdot p_3(l_3^1(e^2), l_3^2(e^2)))$ *are equivalent and both denote the (0-terminal) graph in Fig. 6.6.*

$\mathcal{E}_k$ in Fig. 6.7 is the set of axioms indexed by k. Let $\mathcal{E}_\infty = \cup_k \mathcal{E}_k$ and $\mathcal{E}_{\leq n} = \cup_{k \leq n} \mathcal{E}_k$. We denote the finite application of axioms in a set E is denoted by $=_E$. It is easy to see that each axiom in $\mathcal{E}_\infty$ is sound.

Theorem 6.7 (Soundness) *Let s, t be terms in* $\mathcal{B}_\infty$. *Then s and t are equivalent if* $s =_{\mathcal{E}_\infty} t$.

$$
\begin{array}{rclrl}
p_k(G_1, G_2) & = & p_k(G_2, G_1) & (Commut.) & \text{(AC1)} \\
p_k(p_k(G_1, G_2), G_3) & = & p_k(G_1, p_k(G_2, G_3)) & (Assoc.) & \text{(AC2)} \\
l_k^i \cdot l_{k-1}^{j-1}(G) & = & l_k^j \cdot l_{k-1}^i(G) & 1 \leq i < j \leq k & (l\text{-Com}) \\
p_k(l_k^i(G_1), l_k^i(G_2)) & = & l_k^i(p_{k-1}(G_1, G_2)) & 1 \leq i \leq k & (l\text{-Dist}) \\
r_k \cdot l_k^i(G) & = & l_{k-1}^i \cdot r_{k-1}(G) & 1 \leq i < k & \text{(E1)} \\
r_k \cdot p_k(l_k^k(G_1), G_2) & = & p_{k-1}(G_1, r_k(G_2)) & & \text{(E2)} \\
p_k(G, l_k^k \cdots l_1^1(\mathbf{0})) & = & G & & \text{(E3)} \\
p_2(e^2, e^2) & = & e^2 & & \text{(E4)} \\
\sigma_k^j \cdot l_k^i(G) & = & l_k^i \cdot \sigma_{k-1}^{j-1}(G) & 1 \leq i < j < k & (\sigma\text{1-a}) \\
\sigma_k^i \cdot l_k^i(G) & = & l_k^{i+1}(G) & 1 \leq i < k & (\sigma\text{1-b}) \\
\sigma_k^i \cdot l_k^{i+1}(G) & = & l_k^i(G) & 1 \leq i < k & (\sigma\text{1-c}) \\
\sigma_k^j \cdot l_k^i(G) & = & l_k^i \cdot \sigma_{k-1}^j(G) & 1 < j+1 < i \leq k & (\sigma\text{1-d}) \\
\sigma_2^1(e^2) & = & e^2 & & (\sigma 2) \\
\sigma_k^i(p_k(G_1, G_2)) & = & p_k(\sigma_k^i(G_1), \sigma_k^i(G_2)) & 1 \leq i < k & (\sigma 3) \\
r_k \cdot \sigma_k^i(G) & = & \sigma_{k-1}^i \cdot r_k(G) & 1 \leq i < k-1 & (\sigma 4) \\
r_{k-1} \cdot r_k \cdot \sigma_k^{k-1}(G) & = & r_{k-1} \cdot r_k(G) & & (\sigma 5)
\end{array}
$$

Figure 6.7: Axioms $\mathcal{E}_k$ of the algebraic construction of graphs

Remark For simplicity, we consider axioms only for simple graphs. The extensions for graphs with multiple edges/loops, and/or digraphs are straightforward.

For instance, removal of $(E4)$ in Fig. 6.7 gives the axioms for graphs with multiple edges. By adding a constant l^1 as a 1-terminal graph that consists of the unique terminal and the unique edge from the terminal to the terminal, we obtain the algebraic construction of graphs with loops.

For direct graphs, instead of an edge e^2, we use e^2_+ and e^2_-, where e^2_+ is the directed edge from the first terminal to the second, and e^2_- is opposite. Then, the replacement of $\sigma_2^1(e^2) = e^2$ $(\sigma 2)$ with $\sigma_2^1(e^2_+) = e^2_-$ and $\sigma_2^1(e^2_-) = e^2_+$ lead the axioms for directed graphs.

6.2.2 Complete axiomatization of graphs

Theorem 6.7 and 6.8 show that a(n infinite) set of axioms $\mathcal{E}_\infty$ is sound and complete. The key of the proof of Theorem 6.8 is the existence of a *canonical form*, which will be illustrated in Example 6.13.

Theorem 6.8 (Completeness) *Let s, t be terms in $\mathcal{B}_\infty$. Then $s =_{\mathcal{E}_\infty} t$ if s and t are equivalent.*

We will denote reduction rules by adding the index $_\rightarrow$ or $_\leftarrow$ to the axioms. For instance, $(E1)_\rightarrow$ is a left-to-right reduction rule of the axiom $(E1)$.

Definition 6.9 *For axioms in $\mathcal{E}_\infty$, let TRSs R_1 and R_2 be defined as*

$$\left\{ \begin{array}{lcl} R_1 & = & \{(E1)_\leftarrow, (E2)_\leftarrow, (E2)'_\leftarrow, (l\text{-}Dist)_\leftarrow, (\sigma 3)_\rightarrow, (\sigma 4)_\leftarrow\}, \\ R_2 & = & \{(\sigma 1)_\rightarrow, (\sigma 2)_\rightarrow\}, \end{array} \right.$$

where $(E2)'_\leftarrow$ is $p_{k-1}(r_k(G_1), G_2) \to r_k \cdot p_k(G_1, l_k^k(G_2))$. for each k.

Lemma 6.10 *R_1 and R_2 are terminating over ground terms.*

Proof Let $\delta(t, f)$ be the number of occurrences of a signature f in a term t, and let $\Delta(t, g, f)$ be the sum of all $\delta(s, f)$ where s is a subterm of t such that $root(s) = g$. We define the weight $\omega(t)$ of a term t by

$$\omega(t) = (\omega_{p,r}(t), \omega_{l,r}(t) + \omega_{l,p}(t) + \omega_{\sigma,r}(t) + \omega_{\sigma,p}(t))$$

where

$$\begin{array}{rcl} \omega_{p,r}(t) & = & \Sigma_{i,j,k}\Delta(t, p_k, r_j), \\ \omega_{l,r}(t) & = & \Sigma_{i,j,i',j'}\Delta(t, l_j^i, r_{j'}), \\ \omega_{l,p}(t) & = & \Sigma_{i,j,k}\Delta(t, l_j^i, p_k), \\ \omega_{\sigma,r}(t) & = & \Sigma_{i,j,i',j'}\Delta(t, \sigma_j^i, r_{j'}), \\ \omega_{\sigma,p}(t) & = & \Sigma_{i,j,k}\Delta(t, \sigma_j^i, p_k), \end{array}$$

and define the lexicographic order on the weight. Then, for each reduction of R_1 the weight $\omega(t)$ decreases, and R_1 is **SN**. Similarly, each reduction of R_2 decreases the weight $\omega_{\sigma,l}(t) = \Sigma_{i,j,i',j'}\Delta(t, \sigma_j^i, l_{j'}^{i'})$, and R_2 is **SN**.

Definition 6.11 *A term t is a canonical form if there exist*

- *$R_{n,k}[\ ] = r_{k+1} \cdots r_n[\ ]$,*
- *$P_n[\ , \cdots, \ ]$ consists of p_n,*
- *$L_1[\ ], \cdots, L_m[\ ]$ consist of $\{l_j^i\}$ for $i \le j \le n$,*

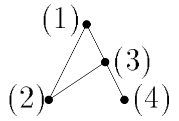
such that $t = R_{n,k}[P_n[L_1[e^2], \cdots, L_m[e^2]]]$ or $t = R_{n,k}[L_1[\mathbf{0}]]$ for $n = |V(\psi(t))|$ and $m = |E(\psi(t))|$.

Lemma 6.12 *For any term t, there exists a canonical form $s \in \mathcal{B}_n$ such that $s =_{\mathcal{E}_{\le n}} t$ where $n = |V(\psi(t))|$.*

Proof We first show that there exists t' in the form $t' = R_{n,k}[P'[L'_1[c_1], \cdots, L'_l[c_l]]]$ with $t =_{\mathcal{E}_{\le n}} t'$ where

- $R_{n,k}[\ ] = r_{k+1} \cdots r_n[\ ]$,
- $P'[\ ]$ consists of p_i's, and
- $L'_1[\ ], \cdots, L'_m[\ ]$ consist of l_j^i's and $\sigma_{k'}^{i'}$'s.
- c_i is either e^2 or $\mathbf{0}$,

$$\begin{aligned}
& r_1 \cdot r_2 \cdot p_2(e^2, r_3 \cdot p_3(l_3^1 \cdot p_2(e^2, \underline{l_2^1 \cdot r_2}(e^2)), l_3^2(e^2))) \\
\to_{R_1}\; & r_1 \cdot r_2 \cdot p_2(e^2, r_3 \cdot p_3(l_3^1 \cdot \underline{p_2(e^2, r_3 \cdot l_3^1(e^2))}), l_3^2(e^2))) \\
\to_{R_1}\; & r_1 \cdot r_2 \cdot p_2(e^2, r_3 \cdot p_3(\underline{l_3^1 \cdot r_3}(p_3(l_3^3(e^2), l_3^1(e^2))), l_3^2(e^2))) \\
\to_{R_1}\; & r_1 \cdot r_2 \cdot p_2(e^2, r_3 \cdot p_3(r_4 \cdot \underline{l_4^1(p_3(l_3^3(e^2), l_3^1(e^2)))}, l_3^2(e^2))) \\
\to_{R_1}\; & r_1 \cdot r_2 \cdot p_2(e^2, r_3 \cdot \underline{p_3(r_4 \cdot p_3(l_4^1 \cdot l_3^3(e^2), l_4^1 \cdot l_3^1(e^2)), l_3^2(e^2))}) \\
\to_{R_1}\; & r_1 \cdot r_2 \cdot \underline{p_2(e^2, r_3 \cdot r_4 \cdot p_4(p_4(l_4^1 \cdot l_3^3(e^2), l_4^1 \cdot l_3^1(e^2)), l_4^4 \cdot l_3^2(e^2)))} \\
\to_{R_1}^{+}\; & \underbrace{r_1 \cdot r_2 \cdot r_3 \cdot r_4}_{R[\,]} \cdot \underbrace{p_4}_{P[\,]}(\underbrace{l_4^4 \cdot l_3^3}_{L_1[\,]}\ \underbrace{(e^2)}, \ p_4(p_4(\underbrace{l_4^4 \cdot l_3^1}_{L_2[\,]}\ \underbrace{(e^2)}, \ \underbrace{l_4^2 \cdot l_3^1}_{L_3[\,]}\ \underbrace{(e^2)}), \ \underbrace{l_4^4 \cdot l_3^2}_{L_4[\,]}\ \underbrace{(e^2)}))
\end{aligned}$$



⇩ Increase the number of terminals

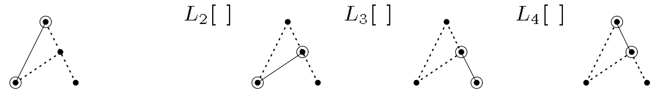


Figure 6.8: Transformation to a canonical form

From Lemma 6.10, t has an R_1-normal form t' of the form $R_{n,k}[P'[L'_1[c_1], \cdots, L'_m[c_m]]]$. Since all vertices in e^2 are terminals and l^i_j, σ^i_j preserves a set of terminals, all vertices of each $L'_m[e^2]$ are terminals. r_i and p_j do not change the number of vertices, thus each p_j in $P'[\ ]$ satisfies $j = n = |V(\psi(t))|$. Further, from Lemma 6.10 each $L'_i[c_i]$ has an R_2-normal form, i.e., a σ^j_k-free term.

If $|E(\psi(t))| = 0$, this means $\psi(t)$ consists of isolated vertices and all c_i's are $\mathbf{0}$. Thus, $L'_i[\ ] = l^k_k \cdots l^1_1[\ ]$ by $(l\text{-}Com)_\leftarrow$ and t is reduced to a canonical form $R_{n,k}[L_1[\mathbf{0}]]$ by (AC1), (AC2), and $(E3)_\rightarrow$.

If $|E(\psi(t))| > 0$, we can sort each $L'_i[\ ]$ by $(l\text{-}Com)_\leftarrow$. Since there exists $c_i = e^2$, we can erase $\mathbf{0}$'s by (AC1), (AC2), and $(E3)_\rightarrow$. Thus we assume $c_i = e^2$ for each i. If $L'_i[c_i]$ and $L'_j[c_j]$ are equal, we can eliminate redundant $L'_i[c_i]$'s by (AC1), (AC2), and $(E4)_\rightarrow$. Then, since each $L'_i[c_i]$ corresponds to an edge in $\psi(t)$ (i.e., the number of $L'_i[c_i]$'s is the number of edges in $\psi(t)$), we obtain a canonical form $t = R_{n,k}[P_n[L_1[e^2], \cdots, L_m[e^2]]]$.

Example 6.13 *Fig. 6.8 shows a transformation to obtain a canonical form of the expression in Example 6.4. The underlined parts correspond to the rewrite steps.*

Sketch of proof of Theorem 6.8 Let s, t be two terms such that $\psi(s)$ and $\psi(t)$ are equivalent such that an isomorphism $\alpha : V(\psi(s)) \to V(\psi(t))$ satisfies the conditions in Definition 6.1. If $|E(\psi(s))| = |E(\psi(t))| = 0$, they have the unique canonical form from Lemma 6.12 and obviously the theorem holds. We assume $|E(\psi(s))| = |E(\psi(t))| > 0$.

From Lemma 6.12, we can assume that both s and t are canonical. Let $n = |V(\psi(s))| = |V(\psi(t))|$, $m = |E(\psi(s))| = |E(\psi(t))|$, $s = R_{n,k}[P_n[L_1[e^2], \cdots, L_m[e^2]]]$, and $t = R_{n,k}[P'_n[L'_1[e^2], \cdots, L'_m[e^2]]]$.

Thus, α can be regarded as the permutation σ on $\{k+1, \cdots, n\}$.

Non-trivial permutation needs at least two elements, so we can assume $k \leq n-2$. Then from $(\sigma 4)$ and $(\sigma 5)$, $r^{k+1}_{k+1} \cdots r^n_n \cdot \sigma^i_n(G) = r^{k+1}_{k+1} \cdots r^n_n(G)$ for $k+1 \leq i \leq n-1$.

Since a permutation over $\{k+1, \cdots, n\}$ is generated by σ_n^i's for $k+1 \leq i \leq n-1$, $r_{k+1}^{k+1} \cdots r_n^n \cdot \sigma(G) = r_{k+1}^{k+1} \cdots r_n^n(G)$. Thus, it is enough to show

$$\sigma(P_n[L_1[e^2], \cdots, L_n[e^2]]) =_{\mathcal{E}_{\leq n}} P'_n[L'_1[e^2], \cdots, L'_n[e^2]]$$

Since $\psi(s)$ and $\psi(t)$ are isomorphic, if there is an edge between the i-th and j-th vertices of $\psi(s)$, there is an edge between the $\alpha(i)$-th and $\alpha(j)$-th vertices of $\psi(t)$, and vice versa. Let $e(n,i,j) = l_n^n \cdots l_{j+1}^{j+1} \cdot l_j^{j-1} \cdots l_{i+2}^{i+1} \cdot l_{i+1}^{i-1} \cdots l_3^1(e^2)$ for $1 \leq i < j \leq n$.

If there is an edge between the i-th and j-th vertices in $\psi(s)$ (resp. the $\alpha(i)$-th and $\alpha(j)$-th vertices in $\psi(t)$) then there uniquely exists $L_k[e^2]$ such that $L_k[e^2] =_{\mathcal{E}_{\leq n}} e(n,i,j)$ (resp. $L'_k[e^2]$ such that $L'_k[e^2] =_{\mathcal{E}_{\leq n}} e(n, \alpha(i), \alpha(j))$.

Since $\sigma(e(n,i,j)) = e(n, \alpha(i), \alpha(j))$, the equation above holds by $(AC1)$, $(AC2)$, $(\sigma 2)$, and $(\sigma 3)$.

6.3 Related Work

Such complete axiomatization was firstly presented in the different algebraic construction [BC87]. Unfortunately, its construction almost squarely grows in size. This is not suitable for a linear time generation. Instead we presented the complete axiomatization of the algebraic construction proposed in [ACPS93], in which the size grows linearly.

Arnborg et al. investigated a canonical form of graph reduction, i.e., a graph holds its property when it reaches to some canonical form by graph reduction. Their method is based on automata theoretic observation. The use of finite congruence makes the canonical forms finite and provides a constructive method for finding canonical forms of graphs with bounded tree width. However, the actual computation of canonical forms is not so easy, and they gave only for small tree widths, such as 2 and 3 [AP92, ACPS93].

There have been proposed several recursive constructions of a graph [FS96, Gib95, Erw97, KL95][2], which were intended to apply program transformational/calculational approaches in functional programming [Fok89, TM95]. For instance, Erwig proposed the construction of directed graphs by

$$\mathit{graph} = \mathit{Empty} \mid (p, v, s) \ \ \& \ \ \mathit{graph}$$

where p is a list of predecessors and s is a list of successors of a vertex v (See Fig. 6.9). Note that the same graph may be denoted by multiple constructions. For example, the directed triangle with the root vertex A can be expressed as

$$\begin{cases} ([\,], A, [B, C]) \ \& \ ([C], B, [\,]) \ \& \ ([\,], C, [\,]) \ \& \ \mathit{Empty}, \\ ([\,], A, [C, B]) \ \& \ ([\,], C, [B]) \ \& \ ([\,], B, [\,]) \ \& \ \mathit{Empty}. \end{cases}$$

Thus, he introduced *active pattern matching* $\underline{\&}$, which traces an expression by converting it with a conditional rule, and demonstrated its flexible expressibility. However, its algebraic structure is not so clear, and the validity of calculational laws is not inconclusive.

[2]Comparison will be found in [Erw97]

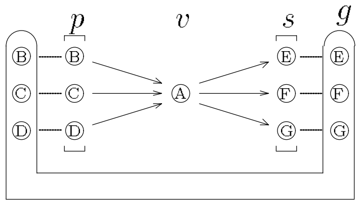


Figure 6.9: Recursive construction of graphs by Erwig

6.4 Future applications on complex structures

The class of graphs with bounded tree width is restrictive; for instance, planar graphs are not (namely, $m \times n$ mesh has tree width $min(m, n)$). However, the class is still useful in practice; there are lots of interesting examples. For instance, in database theory tractable classes of constraint satisfaction problems are found in terms of bounded tree width [Var00].

The more interesting example is a class of control flow graphs of structural programs, namely, tree width of the control flow graph of a `GOTO`-free `C` program is at most 6 [Tho98]. This suggests the possibility to beat the cubic bottleneck of control flow analyses [HM97, Rep98]. The cubic bottleneck is mostly caused from difficulty to solve the variation of graph reachability problems, called CFG-graph reachability. However, the observation that the control flow graph of a program is not an arbitrary graph, but is in a restricted shape constructed by programming language primitives, would improve the complexity of control flow analyses, hopefully to linear time, even at the cost of the restriction to the regular or deterministic CFG reachability .

Of course, this estimation is in theory; for practice, we will face huge constants, say, the tower of exponentials. To reduce constants, our approach is the use of program calculational techniques, such as fusion and tuppling, similar to our result on simple cases [SHTO00, SHTO02]. However, the algebraic construction of graphs (such as tree decomposition) is *not* initial, whereas most of program calculational techniques assume an initial algebra. This demands further investigations, namely,

- complete axiomatization for the algebraic construction of graphs with bounded tree width,
- calculational program transformation for data structures of non-initial algebra.

For the former, we expect that the axiomatization (presented in Section 6.2) of sorts B_k up to some fixed k is complete for graphs with tree width at most $k - 1$.

For the latter, we expect that similar fusion/tuppling transformation techniques is also valid for non-initial algebras as suggested in [Fok96], which must be checked step by step on the proof, such as in [Jür00].

Chapter 7

Concluding Remark

Many automatic support for programming have been realized as their compilers, and many fundamental compiler techniques and their limitations have been clarified. The topics of Part 1 and 2 in this thesis were investigated along with such growth.

If we want more powerful automatic support, we must shift trade-off point under problem-specific restrictions. My recent research, like Part 3, focuses on this direction. Part 3 already discussed *near* future work at the end of each chapter. To conclude the thesis, I would like to discuss more about *far reaching* future works.

The first one would locate in middle distance. Chapter 6 presented the automatic linear time algorithm generation for monadic second order specification over graphs with bounded tree width [BLW87, Cou90, BPT92, SHTO00]. This will grow to the strong tool, but the class of graphs with bounded tree width is restrictive; for instance, planar graphs are not (namely, $m \times n$ mesh has tree width $min(m, n)$).

Recently, Grohe showed automatic (almost) linear time algorithm generation for monadic first-order specification over graphs with locally bounded tree width [FG99, GMn99, Gro00], based on Gaifman's Locality Theorem [Gai82]. This result shows the reasonable trade-off; we need to restrict specification to first order, instead we obtain the larger class of graphs with locally bounded tree width. Note that this class covers planar graphs. Since the reachability is first order definable [AF90], a linear time algorithm finding the shortest path in a planar graph would be automatically generated. [1] This observation also showed us the clear trade-off with the well-known Dijkstra algorithm, which is the most efficient known algorithm for computing the shortest path with complexity $O(e\ log\ n)$ (where e is the number of edges and n is the number of vertices). For this linear time generation, the problem of the constant explosion will also occur as for discussed in Chapter 6. I hope the combination with program calculational techniques would make it practical, too.

The second one, which is truly far reaching future work, is an automatic generation of *probabilistic algorithms.* Of course, there are already lots of work on the individual probabilistic algorithm. However, my view is similar to Part 3; to apply theory of combinatoric and obtain certain automatic support for designing/programming probabilistic algorithms. The possible tool from combinatorics, which I expect, is theory of *random*

[1]As well for a linear time algorithm generation of the cylinder problem presented in [BdM96].

graphs [Spe01, Bol01]. Theory of random graphs introduces us many interesting phenomena; under the assumption that the connecting probability between two vertices is uniform, many global properties follow so-called 0-1 laws, i.e., there exists a threshold. For instance, consider connectivity of a graph. If the connecting probability is smaller than $log\ n/n$ (where n is the number of vertices), then *almost every* graph is *not* connected when $n \to \infty$. In contrast, if the connecting probability is greater than $log\ n/n$, then *almost every* graph is connected. Such threshold always exists for first order definable properties [Spe01].

What I expect is to generate an efficient program whose correctness is *almost* guaranteed. Of course, critical systems, whose correctness must be strictly guaranteed, always exist. However, there will be also strong demand for *almost* correct and efficient programs at low cost. Consider your PC. It runs *almost* correctly, especially if your PC runs on Windows OS.

Probably, these directions will be out of the scope of functional programs. However, among lots of choice of programming languages, still I believe that functional programs will fit to the purpose, because lots of theoretical tools/techniques, such as program transformation, have been developed for them. I hope that this thesis contributes to such development and bridges towards the far reaching future work discussed here.

Bibliography

[AAB$^+$99] P.A. Abdulla, A. Annichini, S. Bensalem, A. Bouajjani, P. Habermehl, and Y. Lakhnech. Verification of infinite-state systems by combining abstraction and reachability analysis. In *Proc. 11th Int. Conf. on Computer Aided Verification, CAV'99*, pages 146–159. Springer-Verlag, 1999. Lecture Notes in Computer Science, Vol. 1633.

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[ACJYK96] P.A. Abdulla, K. Cerans, C. Jonsson, and T. Yih-Kuen. General decidability theorems for infinite-state systems. In *Proc. 10th IEEE Symposium on Logic in Computer Science*, pages 313–321, 1996.

[ACJYK00] P.A. Abdulla, K. Cerans, C. Jonsson, and T. Yih-Kuen. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*, 160(1–2):109–127, 2000.

[ACPS93] S. Arnborg, B. Courcelle, A. Proskurowski, and D. Seese. An algebraic theory of graph reduction. *Journal of the Association for Computing Machinery*, 40(5):1134–1164, 1993.

[Acz78] P. Aczel. A general church-rosser theorem. Preprint, University of Manchester, 1978.

[AF90] M. Ajtai and R. Fagin. Reachability is harder for directed than for undirected graphs. *Journal of Symbolic Logic*, 55(1):113–150, 1990.

[AFFS98] A. Aiken, M. Fähndrich, J.S. Foster, and Z. Su. A toolkit for constructing type- and constraint- based program analyses. In *Proc. 2nd Workshop on Types in Compilation (TIC 1998)*, pages 165–169, 1998.

[AFM$^+$95] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and Wadler P. A call-by-need lambda calculus. In *Proc. 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'95*, pages 233–246. ACM Press, 1995.

[AG97] T. Arts and J. Giesl. Automatically proving termination where simplification orderings fail. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97:*

Theory and Practice of Software Development, pages 261–272. Springer-Verlag, 1997. Lecture Notes in Computer Science, Vol.1214.

[AG00] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1–2):133–178, 2000.

[AH87] S. Abramsky and C. Hankin, editors. *Abstract interpretation of declarative languages.* Ellis Horwood Limited, 1987.

[Aka93] Y. Akama. On mints' reduction for ccc-calculus. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculus and Applications*, pages 1–12. Springer-Verlag, 1993. Lecture Notes in Computer Science, Vol. 664.

[AKW95] A. Aiken, D. Kozen, and E.L. Wimmers. Decidability of systems of set constraints with negative constraints. *Information and Computation*, 122(1):30–44, 1995.

[AN00] P.A. Abdulla and A. Nyl'en. Better is better than well: On efficient verification of infinite-state systems. In *Proc. 14th IEEE Symposium on Logic in Computer Science*, pages 132–140, 2000.

[AP92] S. Arnborg and A. Proskurowski. Canonical representations of partial 2- and 3-trees. *BIT*, 32:197–214, 1992.

[APS90] S. Arnborg, A. Proskurowski, and D. Seese. Monadic second order logic, tree automata and forbidden minors. In *Proc. 4th Workshop in Computer Science Logic, CSL'90*, pages 1–16. Springer-Verlag, 1990. Lecture Notes in Computer Science, Vol. 533.

[Arn85] A. Arnold. A syntactic congruence for rational ω-languages. *Theoretical Computer Science*, 39:333–335, 1985.

[AU77] A.V. Aho and J.D. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1977.

[B$^+$97] F. Bueno et al. On the role of semantic approximations in validation and diagnosis of constraint logic programs. In *Proc. AADEBUG '97*, pages 155–169, 1997.

[Bar84] H.P. Barendregt. *The Lambda Calculus, its Syntax and Semantics.* North-Holland, Amsterdam, 2nd edition, 1984.

[BBKH76] H. P. Barendregt, J. Bergstra, J. W. Klop, and Volken H. Some notes on lambda-reduction. Technical Report 22, Department of mathematics, University of Utrecht, 1976. pp. 13–53 *in* Degrees, reductions, and representability in the lambda calculus.

[BC87] M. Bauderon and B. Courcelle. Graph expressions and graph rewritings. *Mathematical System Theory*, 20:83–127, 1987.

[BdM96] R. Bird and O. de Moor. *Algebra of Programming.* Prentice Hall, 1996.

[Ben84] J.L. Bentley. Programming pearls: Algorithm design techniques. *Communications of the ACM*, 27(9):865–871, 1984.

[Ben93] P.N. Benton. Strictness properties of lazy algebraic datatypes. In *Static Analysis*, pages 206–217. Springer-Verlag, 1993. Lecture Notes in Computer Science, Vol. 724.

[BGW93] L. Bachmair, H. Ganzinger, and U. Waldmann. Set constraints are the monadic class. In *Proc. 8th IEEE Symposium on Logic in Computer Science*, pages 75–83, 1993.

[BI94] C. Böhm and B Intrigila. The ant-lion paradigm for strong normalization. *Inform. and Comp.*, 114(1):30–49, 1994.

[BK82] J. A. Bergstra and J. W. Klop. Strong normalization and perpetual reductions in the lambda calculus. *J. Inform. Process. Cybernet*, 18:403–417, 1982.

[BK86] J. A. Bergstra and J. W. Klop. Conditional rewrite rules: confluence and termination. *J. Comp. Sys. Sci.*, 32(3):323–362, 1986.

[Blo86] P. Bloss, B. and Hudak. Variations on strictness analysis. In *Proc. ACM Conference on LISP and Functional Programming*, pages 132–142. ACM Press, 1986.

[BLW87] M.W. Bern, E.L. Lawler, and A.L. Wong. Linear-time computation of optimal subgraphs of decomposable graphs. *Journal of Algorithms*, 8:216–235, 1987.

[Bol01] B. Bollobás. *Random Graphs.* Academic Press, 2001. Second edition.

[Bon00] E. Bonelli. Perpetuality in a named lambda calculus with explicit substitutions. to appear in *Mathematical Structures in Computer Science*, 2000.

[Bou85] G. Boudol. Computational semantics of term rewriting systems. In M. Nivat and J. C. Reynolds, editors, *Algebraic methods in semantics*, pages 169–236. Cambridge University Press, Cambridge, UK, 1985.

[Bou93a] F. Bourdoncle. Abstract debugging of higher-order imperative programs. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'93*, pages 46–55. ACM Press, 1993.

[Bou93b] F. Bourdoncle. Assertion-based debugging of imperative programs by abstract interpretation. In *4th ESEC*, pages 501–516. Springer-Verlag, 1993. Lecture Notes in Computer Science, Vol. 717.

[BPT92] R.B. Borie, R.G. Parker, and C.A. Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, 7:555–581, 1992.

[BRS99] S. Brin, R. Rastogi, and K. Shim. Mining optimized gain rules for numeric attributes. In *Proc. 5th ACM SIGKDD International Conference on Knowledge and Data Mining, KDD 1999*, pages 135–144. ACM Press, 1999.

[Bur90] G.L. Burn. A relationship between abstract interpretation and projection analysis. In *Proc. 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 151–156. ACM Press, 1990.

[Bur91] G.L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation.* The MIT Press, 1991.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[CC92] P. Cousot and R. Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *Programming Language Implementation and Logic Programming*, pages 269–295. Springer-Verlag, 1992. Lecture Notes in Computer Science, Vol. 631.

[CF93] T. Coquand and D. Fridlender. A proof of Higman's lemma by structural induction, November 1993. available at `http://www.md.chalmers.se/~coquand/intuitionism.html`.

[CFW91] A. Cortesi, G. Filé, and W. Winsborough. *Prop* revisited: Propositional formula as abstract domain for groundness analysis. In *Proc. 6th IEEE Symposium on Logic in Computer Science*, pages 322–327, 1991.

[Che81] P. Chew. Unique normal forms in term rewriting systems with repeated variables. In *Proc. 13th ACM Symposium on Theory of Computing*, pages 7–18. ACM Press, 1981.

[Chu41] A. Church. *The Calculi of Lambda-Conversion.* Princeton Univ. Press, 1941.

[CK00] W.-N. Chin and S.-C. Khoo. Calculating sized types. In *Proc. 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantic-Based Program Manipulation, PEPM'00*, pages 62–72. ACM Press, 2000.

[CLCR96] A. Cortesi, B. Le Charlier, and S. Rossi. Specification-based automatic verification of Prolog programs. In *Logic Program Synthesis and Transformation*, pages 38–57. Springer-Verlag, 1996. Lecture Notes in Computer Science, Vol. 1207.

[CLMV96] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of logic programs by abstract diagnosis. In M. Dams, editor, *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop*, Lecture Notes in Computer Science, Vol. 1192, pages 22–50. Springer-Verlag, 1996.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms.* The MIT Electrical Engineering and Computer Science Series. MIT Press, 1990.

[CM01] B. Courcelle and J.A. Makowsky. Fusion in relational structures and the verification of monadic second-order properties. *Mathematical Structures in Computer Science*, 2001. to appear.

[Cou90] B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 5, pages 194–242. Elsevier Science Publishers, 1990.

[CPJ85] C. Clack and S.L. Peyton Jones. Strictness analysis – a practical approach. In *Functional Programming Languages and Computer Architecture*, pages 35–49. Springer-Verlag, 1985. Lecture Notes in Computer Science, Vol. 201.

[Der82] N. Dershowitz. Ordering for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[DF85] D. Detlefs and R. Forgaard. A procedure for automatically proving the termination of a set of rewrite rules. In *Rewriting Techniques and Applications*, pages 255–270. Springer-Verlag, 1985. Lecture Notes in Computer Science, Vol. 202.

[DF95] R.G. Downey and M.R. Fellows. Fixed-parameter tractability and completeness I: Basic results. *SIAM Journal Computing*, 24:873–921, 1995.

[DG93] P. De Groote. The conservation theorem revisited. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculus and Applications*, pages 163–178. Springer-Verlag, 1993. Lecture Notes in Computer Science, Vol. 664.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 243–320. Elsevier Science Publishers, 1990.

[dMS99] O. de Moor and G. Sittampalam. Generic program transformation. In J.N. Oliveria, editor, *3rd International Summer School on Advanced Functional Programming*, pages 116–149. Springer-Verlag, 1999. Lecture Notes in Computer Science, Vol. 1608.

[DOG88] N. Dershowitz, M. Okada, and Sivakumar G. Canonical conditional rewrite systems. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 538–549. Springer-Verlag, 1988. Lecture Notes in Computer Science, Vol. 310.

[DV87] R. C. De Vrijer. *Surjective pairing and strong normalization: two themes in lambda calculus*. PhD thesis, Universiteit van Amsterdam, 1987.

[DW90] K. Davis and P. Wadler. Backwards strictness analysis: Proved and improved. In Davis K. and Hughes J., editors, *in Functional Programming: Proc. 1989 Glasgow Workshop*, pages 12–30. Springer-Verlag, 1990.

[DW91] K. Davis and P. Wadler. Strictness analysis in 4d. In Hutton G. Peyton Jones S.L. and Holst C.K., editors, *in Functional Programming: Proc. 1990 Glasgow Workshop*, pages 23–43. Springer-Verlag, 1991.

[Dyb91] P. Dybjer. Inverse image analysis generalizes strictness analysis. *Information and Computation*, 90(2):194–216, 1991.

[EHR83] A. Ehrenfeucht, D. Hausser, and G. Rozenberg. On regularity of context-free languages. *Theoretical Computer Science*, 27:311–332, 1983.

[EM91] C. Ernoult and A. Mycroft. Uniform ideals and strictness analysis. In *Automata, Languages and Programming*, pages 47–59. Springer-Verlag, 1991. Lecture Notes in Computer Science, Vol. 510.

[Erw97] M. Erwig. Functional progamming with graphs. In *Proc. 2nd ACM SIGPLAN International Conference on Functional Programming*, pages 52–64. ACM Press, 1997.

[FFSA98] M. Fähndrich, J.S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constrint graphs. In *Proc. ACM SIGPLAN '98 Symposium on Principles of Programming Language Design and Implementation*, pages 85–96. ACM Press, 1998.

[FG99] M. Frick and M. Grohe. Deciding first-order properties of locally tree-decomposable structures. In J. Wiedermann, P. van Emde Boas, and M. Nielson, editors, *Automata, Languages and Programming*, pages 532–543. Springer-Verlag, 1999. Lecture Notes in Computer Science, Vol. 1644.

[FL88] M.R. Fellows and M.A. Langston. Nonconstructive tools for proving polynomial-time decidability. *Journal of the ACM*, 35(3):727–739, 1988.

[Flu01] J. Flum. Tree-decompositions and the model-checking problem. *Bulletin of the Europian Association for Theoretical Computer Science*, 73:78–98, 2001.

[FMMT96a] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Interval finding and its application to data mining. In *Proc. ISSAC96 (International Symposium on Algorithms and Computation)*, pages 55–64. Springer-Verlag, 1996. Lecture Notes in Computer Science, Vol. 1178.

[FMMT96b] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Mining optimized association rules for numeric attributes. In *Proc. 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS'96*, pages 182–191. ACM Press, 1996.

[Fok89] M. Fokkinga. Tuppling and mutumorphisms. *Squiggolist*, 1(4), 1989.

[Fok96] Maarten M. Fokkinga. Datatype laws without signatures. *Mathematical Structures in Computer Science*, 6:1–32, 1996.

[Fri96] D. Fridlender. Higman's lemma in type theory. In E. Gimnez and C. P.-Mohring, editors, *Types for Proofs and Programs, TYPES'96*, pages 112–133. Springer-Verlag, 1996. Lecture Notes in Computer Science, Vol. 1512.

[Fri99] M. Frigo. A fast fourier transform compiler. In *Proc. ACM Conference on Programming Language Design and Implementations*, pages 169–180. ACM Press, 1999.

[FS96] L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'96*, pages 284–294. ACM Press, 1996.

[FS98] A. Finkel and Ph. Schnoebelen. Fundamental structures in well-structured infinite transition systems. In *Proc. 3rd Latin American Theoretical Informatics Symposium, LATIN'98*, pages 102–118. Springer-Verlag, 1998. Lecture Notes in Computer Science, Vol. 1380.

[Gai82] H. Gaifman. On local and non-local properties. In *Proc. of the Herbrand Symposium, Logic Colloquium '81*. North-Holland, 1982.

[Ges96] A. Geser. A proof of Higman's lemma by open induction. Technical Report MIP-9606, Passau University, April 1996.

[Gib95] J. Gibbons. An initial-algebra approach to directed acyclic graphs. In B. Moller, editor, *Mathematics of Program Construction, MPC'95*, pages 282–303. Springer-Verlag, 1995. Lecture Notes in Computer Science, Vol. 947.

[Gis97] J. Gisel. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19:1–29, 1997.

[GMn99] M. Grohe and J. Mariño. Definability and descriptive complexity on databases of bounded tree-width. In C. Beeri and P. Buneman, editors, *Database Theory - ICDT'99*, pages 70–82. Springer-Verlag, 1999. Lecture Notes in Computer Science, Vol. 1540.

[Gra96] B. Gramlich. *Termination and confluence properties of structured rewrite systems.* PhD thesis, Universität Kaiserslautern, 1996.

[Gri90] D. Gries. The maximum-segment sum problem. In E.W. Dijkstra, editor, *Formal Develeopement of Programs and Proofs*. Addison-Wesley, 1990.

[Gro00] M. Grohe. Isomorphism testing for embeddable graphs through definability. In *Proc. 32nd ACM Symposium on Theory of Computing*. ACM Press, 2000.

[Gro01] M. Grohe. The parameterized complexity of database queries. In *Proc. 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'01*. ACM Press, 2001.

[GTT93] R. Gilleron, S. Tison, and M. Tommasi. Solving systems of set constraints using tree automata. In *STACS 93*, pages 505–514. Springer-Verlag, 1993. Lecture Notes in Computer Science, Vol. 665.

[Gup92] A. Gupta. A constructive proof that trees are well-quasi-ordered under minors. In A. Nerode and M. Taitslin, editors, *Logical foundations of computer science - Tver'92*, pages 174–185. Springer-Verlag, 1992. Lecture Notes in Computer Science, Vol. 620.

[Hal90] C. Hall. Using lazy evaluation to find fixpoints in infinite domains. In B.D.Shriver G.David, R.T.Boute, editor, *Declarative Systems*, pages 85–98. North-Holland, 1990.

[Hei93] N. Heintze. Set constraints in program analysis. In *Proc. International Symposium on Logic Programming*, 1993.

[Hei94] N. Heintze. Set-based analysis of ml programs. In *Proc. 1994 ACM Conference on LISP and Functional Programming*. ACM Press, 1994.

[Hen80] P. Henderson. *Functional Programming: Application and Implementation*. Prentice-Hall, 1980.

[Hig52] G. Higman. Ordering by divisibility in abstract algebras. *Proc. London Mathematical Society*, 2:326–336, 1952.

[HITT97] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tuppling calculation eliminates multiple data traversals. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 164–175. ACM Press, 1997.

[HJ90] N. Heintze and J. Jaffer. A decision procedure for a class of set constraints. In *Proc. 5th IEEE Symposium on Logic in Computer Science*, pages 42–51, 1990.

[HL91] G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic, Essays in Honor of Alan Robinson*, pages 394–443. MIT Press, Cambridge, MA., 1991.

[HL93] F. Honsell and M. Lenisa. Some results on the full abstraction problem for restricted lambda calculi. In A. M. Borzyszkowski and S. Sokolowski, editors, *Mathematical Foundations of Computer Science 1993*, pages 84–104. Springer-Verlag, 1993. Lecture Notes in Computer Science, Vol. 711.

[HL94] J. Hughes and J. Launchbury. Reversing abstract interpretations. *Science of Computer Science*, 22:307–326, 1994.

[HL99] F. Honsell and M. Lenisa. Semantical analysis of perpetual strategies in λ-calculus. *Theoretcal Computer Science*, 212(1–2):183–209, 1999.

[HLM94] C. Hankin and D. Le Métayer. A type-based framework for program analsysis. In *Static Analysis*, pages 380–394. Springer-Verlag, 1994. Lecture Notes in Computer Science, Vol. 864.

[HM97] N. Heintze and D. McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Proc. 12th Annual IEEE Symposium on Logic in Computer Science*, pages 342–351, 1997.

[HP96] M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. In H. Ganzinger, editor, *Rewriting Techniques and Applications*, pages 138–152. Springer-Verlag, 1996. Lecture Notes in Computer Science, Vol. 1103.

[HPS96] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems usign sized types. In *Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'96*, pages 410–423. ACM Press, 1996.

[Hun91] S. Hunt. PERs generalize projections for strictness analysis (extended abstract). In Hutton G. Peyton Jones S.L. and Holst C.K., editors, *in Functional Programming: Proc. 1990 Glasgow Workshop*, pages 114–125. Springer-Verlag, 1991.

[HY86] P. Hudak and J. Young. Higher-order strictness analysis in untyped lambda calculus. In *Proc. 13th ACM Symposium on Principles of Programming languages*, pages 97–109. ACM Press, 1986.

[JA84] I. Jachner and V. K. Agarwal. Data-flow anomaly detection. *IEEE Trans. Software Eng.*, SE-10(4):432–437, 1984.

[Jag89] N. Jagger. An inductive approach to finding fixpoints in abstract interpretation. In *Proc. 4th IEEE Region 10 International Conference, Information Technologies for the 90's (TENCON '89)*, pages 1059–1064, 1989.

[Jan99] P. Jančar. A note on well quasi-orderings for powersets. *Information Processing Letters*, 72(5–6):155–160, 1999.

[Jen94] T.P. Jensen. Abstract interpretation over algebraic data types. In *Proc. International Conference on Computer Languages*, pages 265–276. IEEE, 1994.

[Jür00] C. Jürgensen. A formalization of hylomorphism based deforestation with an application to an extended typed λ-calculus. Technical Report TUD-FI00-13, Technische Universität Dresden, Fakultät Informatik, D-01062 Dresden, Germany, November 2000.

[Kar85] M. Karr. "delayability" in proofs of strong normalizability in the typed λ-calculus. In H. Ehrig, C. Floyd, M. Nivat, and J. Tatcher, editors, *Mathematical Foundations of Computer Software*, pages 208–222. Springer-Verlag, 1985. Lecture Notes in Computer Science, Vol. 185.

[Kha90] Z. Khasidashvili. β-reductions and β-developments of λ-terms with the least number of steps. In P. Martin-Löf and G. Mints, editors, *COLOG-88*, pages 105–111. Springer-Verlag, 1990. Lecture Notes in Computer Science, Vol. 417.

[Kha92] Z. Khasidashvili. The church-rosser theorem in orthogonal combinatory reduction systems. Technical Report 1825, INRIA Rocquencourt, 1992.

[Kha93] Z. Khasidashvili. Optimal normalization in orthogonal term rewriting systems. In C. Kirchner, editor, *Rewriting Techniques and Applications*, pages 243–258. Springer-Verlag, 1993. Lecture Notes in Computer Science, Vol. 690.

[Kha94a] Z. Khasidashvili. The longest perpetual reductions in orthogonal expression reduction systems. In A. Nerode and Yu. V. Matiyasevich, editors, *Logical Foundations of Computer Science*, pages 191–203. Springer-Verlag, 1994. Lecture Notes in Computer Science, Vol. 813.

[Kha94b] Z. Khasidashvili. On higher order recursive program schemes. In S. Tison, editor, *Trees in Algebra and Programming - CAAP'94*, pages 172–186. Springer-Verlag, 1994. Lecture Notes in Computer Science, Vol. 787.

[Kha94c] Z. Khasidashvili. Perpetuality and strong normalization in orthogonal term rewriting systems. In P. Enjalbert, E. W. Mayr, and K. W. Wagner, editors, *STACS 94*, pages 163–174. Springer-Verlag, 1994. Lecture Notes in Computer Science, Vol. 775.

[Kha01] Z. Khasidashvili. On longest perpetual reductions in orthogonal expression reduction systems. *Theoretcal Computer Science*, 266(1–2):737–772, 2001.

[KL95] D.J. King and J. Launchberry. Structuring depth-first search algorithms in Haskell. In *Proc. 1995 ACM SIGPLAN International Symposium on Principles of Programming Languages, POPL '95*, pages 344–354. ACM Press, 1995.

[Klo80] J. W. Klop. *Combinatory Reduction Systems.* PhD thesis, Utrecht University, 1980. CWI Tracts no. 127, Amsterdam.

[Klo92] J. W. Klop. Term rewriting systems. In D. Gabbay S. Abramsky and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, 1992.

[KOvO01a] Z. Khasidashvili, M. Ogawa, and V. van Oostrom. Perpetuality and uniform normalization in orthogonal rewrite systems. *Information and Computation*, 164:118–151, 2001.

[KOvO01b] Z. Khasidashvili, M. Ogawa, and V. van Oostrom. Uniform normalization beyond orthogonality. In A. Middeldorp, editor, *Rewrting Techniques and Applications*, pages 122–136. Springer-Verlag, 2001. Lecture Notes in Computer Science, Vol. 2051.

[KvO95a] Z. Khasidashvili and V. van Oostrom. Context-sensitive conditional expression reduction systems. In A. Corradini and U. Montanari, editors, *SEGRAGRA 1995*, pages 394–443. Elsevier Science Publishers B.V., Amsterdam, 1995. Electronic Notes in Theoretical Computer Science, Vol.2.

[KvO95b] Z. Khasidashvili and V. van Oostrom. Context-sensitive conditional rewrite systems. Technical Report SYS–C95–06, University of East Anglia, 1995.

[KvOvR93] J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121(1–2):279–308, 1993.

[KW95a] A. J. Kfoury and J. Wells. Addendum to 'new notions of reduction and non-semantic proofs of β-strong normalization in typed λ-calculi. Technical Report 95–007, Boston University Computer Science Department, 1995.

[KW95b] A. J. Kfoury and J. Wells. New notions of reduction and non-semantic proofs of strong β normalization in typed λ-calculi. In D. Kozen, editor, *Logic in Computer Science*, pages 311–321. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[Lau91] J. Launchbury. Strictness and binding-time analyses: Two for the price of one. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'91*, pages 80–91. ACM Press, 1991.

[Lav78] R. Laver. Better-quasi-orderings and a class of trees. In *Studies in foundations and combinatorics, advances in mathematics supplementary studies*, volume 1, pages 31–48. Academic Press, 1978.

[LCLRC97] B. Le Charlier, C. Leclère, S. Rossi, and A. Cortesi. Automatic verification of behavioral properties of Prolog programs. In *Advances in Computer Science - ASIAN '97*, pages 225–237. Springer-Verlag, 1997. Lecture Notes in Computer Science, Vol. 1345.

[Len97a] M. Lenisa. Semantic techniques for deriving coinductive characterizations of observational equivalences for λ-calculi. In P. De Groote and J. R. Hindley, editors, *Typed Lambda Calculus and Applications*, pages 248–266. Springer-Verlag, 1997. Lecture Notes in Computer Science, Vol. 1210.

[Len97b] M. Lenisa. A uniform syntactical method for proving coinduction principles in λ-calculi. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development*, pages 309–320. Springer-Verlag, 1997. Lecture Notes in Computer Science, Vol. 1214.

[Lév78] J.-J. Lévy. *Réductions correctes et optimales dans le lambda-calcul.* PhD thesis, Université Paris VII, 1978.

[Lév80] J.-J. Lévy. Optimal reductions in the lambda-calculus. In J.R. Hindley and J.P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism*, pages 159–192. Academic Press Limited, London, 1980.

[LM95] D. Le Métayer. Proving properties of programs defined over recursive data structures. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantic-Based Program Manipulation, PEPM'95*, pages 88–99. ACM Press, 1995.

[LP00] X. Leroy and F. Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Programming Languages and Systems*, 22(2):340–377, 2000.

[LS93] C. A. Loría-Sáenz. *A theoretical framework for reasoning about program construction based on extensions of rewrite systems.* PhD thesis, Universität Kaiserslautern, 1993.

[LV98] G. Levi and P. Volpe. Derivation of proof methods by abstract interpretation. In *Algebraic and Logic Programming*, pages 102–117. Springer-Verlag, 1998. Lecture Notes in Computer Science, Vol. 1490.

[Mel96] P.-A. Melliès. *Description abstraite des systèmes de réécriture.* PhD thesis, Université Paris VII, 1996.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. FPCA '91 Conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Springer-Verlag, 1991. Lecture Notes in Computer Science, Vol. 523.

[MK84] P. Mishra and A.M. Keller. Static inference of properties of applicative programs. In *Proc. 11th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 235–244, 1984.

[MN98] R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192:3–29, 1998.

[MNS99] P. Møller Neergaard and M. H. Sørensen. Conservation and uniform normalization in lambda calculi with erasing reductions. Submitted for publication, 1999.

[MO01] K. Mano and M. Ogawa. Unique normal form property of compatible term rewriting systems - a new proof of chew's theorem -. *Theoretical Computer Science*, 258(1–2):169–208, 2001.

[MR90] C.R. Murthy and J.R. Russell. A constructive proof of Higman's lemma. In *Proc. 5th IEEE Symposium on Logic in Computer Science*, pages 257–267, 1990.

[MR00] D. Melski and T.W. Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248(1–2):29–98, 2000.

[Myc80] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *International Symposium on Programming*, pages 269–281. Springer-Verlag, 1980. Lecture Notes in Computer Science, Vol. 83.

[Ned73] R. P. Nederpelt. *Strong normalization for a typed lambda-calculus with lambda structured types.* PhD thesis, Technische Hogeschool Eindhoven, 1973.

[Nie86] F. Nielson. A bibliography on abstract interpretation. *ACM SIGPLAN Notices*, 21(5):31–38, 1986.

[Nip93] T. Nipkow. Orthogonal higher-order rewrite systems are confluent. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculus and Applications*, pages 306–317. Springer-Verlag, 1993. Lecture Notes in Computer Science, Vol. 664.

[NNH99] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[NP88] N.Robertson and P.D.Seymour. Graph minors xx. wagner's conjecture, 1988. Manuscript.

[NW63] C.ST.J.A. Nash-Williams. On well-quasi-ordering finite trees. *Proc. Cambridge Phil. Soc.*, 59:833–835, 1963.

[NW65] C.ST.J.A. Nash-Williams. On well-quasi-ordering infinite trees. *Proc. Cambridge Phil. Soc.*, 61:697–720, 1965.

[NZ79] N.Dershowitz and Z.Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[O'D77] M. J. O'Donnell. *Computing in Systems Described by Equations.* Springer-Verlag, 1977. Lecture Notes in Computer Science, Vol. 58.

[Oga99] M. Ogawa. Automatic verification based on abstract interpretation. In *Functional and Logic Programming*, pages 131–146. Springer-Verlag, 1999. Lecture Notes in Computer Science, Vol. 1722.

[Ono88] S. Ono. Relationships among strictness-related analyses for applicative languages. In K. Fuchi and L. Kott, editors, *Proc. of the Second Franco-Japanese Symposium on Programming of Future Generation Computers*, pages 257–283. North-Holland, 1988.

[OO90] K.M. Olender and L.J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Trans. on Software Engineering*, 16(3):268–280, 1990.

[OO91a] M. Ogawa and S. Ono. Deriving inductive properties of recursive programs based on least-fixpoint computation. *Journal of Information Processing*, 32(7):914–923, 1991. *in Japanese.*

[OO91b] M. Ogawa and S. Ono. Transformation of strictness-related analyses based on abstract interpretation. *IEICE Trans.*, E 74(2):406–416, 1991. Conference Version: *Proc. International Conference on Fifth Generation Computer Systems 1988 (FGCS'88)*, pp. 430–438, 1988.

[OTA84] S. Ono, N. Takahashi, and M. Amamiya. Non-strict partial computation with a dataflow machine. In *Proc. 6th RIMS Symposium on Mathematical Methods in Software Science and Engineering (SSE '84)*, pages 196–229, 1984. RIMS Kyoto Univ. TR.547.

[OTA86] S. Ono, N. Takahashi, and M. Amamiya. Optimized demand-driven evaluation of functional programs on a dataflow machine. In *Proc. 15th IEEE International Conference on Parallel Processing*, pages 421–428, 1986.

[P.92] Barendregt H. P. Lambda calculi with types. In D. Gabbay S. Abramsky and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992.

[Pau96] L.C. Paulson. *ML for the working programmer.* Cambridge University Press, 2nd edition, 1996.

[PJ87] S. L. Peyton Jones. *The implementation of functional programming languages.* Prentice-Hall, 1987.

[Pkh77] Sh. Pkhakadze. Some problems of the notation theory. Proc. I. Vekua Institute of Applied Mathematics of Tbilisi State University, Tbilisi, 1977. in Russian.

[Pla93] D. A. Plaisted. Polynomial time termination and constraint satisfaction tests. In C. Kirchner, editor, *Rewriting Techniques and Applications*, pages 405–420. Springer-Verlag, 1993. Lecture Notes in Computer Science, Vol. 690.

[Plo75] G. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo76] G.D. Plotkin. A powerdomain construction. *SIAM Journal Computing*, 5(3):452–487, 1976.

[PR99] L. Perković and B. Reed. An improved algorithm for finding tree decompositions of small width. In P. Widmayer, G. Neyer, and S. Eidenbenz, editors, *WG'99*, pages 148–154. Springer-Verlag, 1999. Lecture Notes in Computer Science, Vol. 1665.

[Rad54] R. Rado. Partial well-ordering of sets of vectors. *Mathematika*, 1:89–95, 1954.

[Rep98] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40:701–726, 1998.

[RS86] N. Robertson and P.D. Seymour. Graph minors ii. algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986.

[RS93] F. Richman and G. Stolzenberg. Well quasi-ordered sets. *Advances in Mathematics*, 97:145–153, 1993.

[RS95] N. Robertson and P.D. Seymour. Graph minors xiii. the disjoint path problem. *Journal of Combinatorial Theory Series B*, 63:65–110, 1995.

[Sam69] J.E. Sammet. *Programming Languages: History and Fundamentals.* Prentice-Hall, 1969.

[Sch98] D.A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proc. 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'98*, pages 38–48. ACM Press, 1998.

[Seg00] C. Seger. Combining fucntioncal programing and hardware verification. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming*, page 244. ACM Press, 2000.

[SF89] G. Springer and D.P. Friedman. *Scheme and the Art of Programming.* The MIT Press, 1989.

[SFA00] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: reducing redundancies in inclusion constraint graphs. In *Proc. 27th ACM SIGPLAN Symposium on Principles of Programming Language Design and Implementation*, pages 81–95. ACM Press, 2000.

[SHT01] I. Sasano, Z. Hu, and M. Takeichi. Generation of efficient programs for maximum marking problems. In *Proc. ACM SIGPLAN Workshop on Semantics, Applications, and Implementation of Program Generation, SAIG'01*, pages 72–91. Springer-Verlag, 2001. Lecture Notes in Computer Science, Vol.2196.

[SHTO00] I. Sasano, Z. Hu, M. Takeichi, and M. Ogawa. Make it practical: A generic linear-time algorithms for solving maximum-weightsum problems. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming*, pages 137–149. ACM Press, 2000.

[SHTO02] I. Sasano, Z. Hu, M. Takeichi, and M. Ogawa. Derivation of linear time algorithm for mining optimized gain association rules. **コンピュータソフトウェア**, 2002. 掲載予定.

[Sim85] S.G. Simpson. Bqo Theory and Fraïssé's Conjecture. In *Recursive Aspects of Descriptive Set Theory*, volume 11 of *Oxford Logic Guides*, chapter 9. Oxford University Press, 1985.

[Sim88] S.G. Simpson. Ordinal numbers and the hilbert basis theorem. *Journal of Symbolic Logic*, 53(3):961–974, 1988.

[Smy78] M.B. Smyth. Power domains. *Journal of Computer and System Science*, 16:23–36, 1978.

[Sør97a] M. H. Sørensen. *Normalization in λ-calculus and type theory.* PhD thesis, Københavns Universitet, 1997.

[Sør97b] M. H. Sørensen. Strong normalization from weak normalization in typed λ-calculi. *Inform. and Comp.*, 133(1):35–71, 1997.

[Sør98] M. H. Sørensen. Properties of infinite reduction paths in untyped λ-calculus. In J. Ginzburg, Z. Khasidashvili, J.J. Lévy, C. Vogel, and E. Vallduví, editors, *Tbilisi Symposium on Logic, Language and Computation, Selected papers*, pages 353–367. SiLLI Publications, CSLI, Stanford, 1998.

[Spe01] J. Spencer. *The Strange Logic of Random Graphs.* Springer-Verlag, 2001.

[Ste91] B. Steffen. Data flow analysis as model checking. In *Theoretical Aspects of Computer Science*, pages 346–364. Springer-Verlag, 1991. Lecture Notes in Computer Science, Vol. 526.

[Tho90a] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 4, pages 133–192. Elsevier Science Publishers, 1990.

[Tho90b] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 4, pages 133–192. Elsevier Science Publishers, 1990.

[Tho98] M. Thorup. All structured programs have small tree width and good register allocation. *Information and Computation*, 142:159–181, 1998.

[TM95] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proc. FPCA '95 SIGPLAN Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, 1995.

[Toy89] Y. Toyama. Membership conditional term rewriting systems. *Trans. IEICE Japan*, E72:1224–1229, 1989. previously appeared in Conditional Term Rewriting Systems (S. Kaplan and J.-P. Jouannaud, Eds.), pp. 128–141, 1988, Lecture Notes in Computer Science, Vol. 308, Springer-Verlag.

[TS96] T. A Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge Univ. Press, Cambridge, UK, 1996. Cambridge Tracts in Theoretical Computer Science, Vol. 43.

[van97] R. van der Meyden. The complexity of querying indefinite data about linearly ordered domains. *Journal of Computer and System Science*, 54(1):113–135, 1997. Previously presented at 11th ACM Symposium on Principles of Database Systems, pp.331–345, 1992.

[Var00] M.Y. Vardi. Constraint satisfaction and database theory: a tutorial. In *Proc. 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'00*, pages 76–85. ACM Press, 2000.

[Vel00] W. Veldman. An intuitionistic proof of Kruskal's theorem. Technical Report 17, Department of Mathematics, University of Nijmegen, April 2000.

[VO94] V. Van Oostrom. *Confluence for abstract and higher-order rewriting*. PhD thesis, Vrije Universiteit, Amsterdam, 1994.

[VO96] V. Van Oostrom. Higher-order families. In H. Ganzinger, editor, *Rewriting Techniques and Applications*, pages 392–407. Springer-Verlag, 1996. Lecture Notes in Computer Science, Vol. 1103.

[VO97] V. Van Oostrom. Finite family developments. In H. Comon, editor, *Rewriting Techniques and Applications*, pages 308–322. Springer-Verlag, 1997. Lecture Notes in Computer Science, Vol. 1232.

[VOvR94] V. Van Oostrom and F. van Raamsdonk. Weak orthogonality implies confluence: the higher-order case. In A. Nerode and Yu. V. Matiyasevich, editors, *Logical Foundations of Computer Science*, pages 379–392. Springer-Verlag, 1994. Lecture Notes in Computer Science, Vol. 813.

[VR96] F. Van Raamsdonk. *Confluence and normalisation for higher-order rewriting*. PhD thesis, Vrije Universiteit, Amsterdam, 1996.

[VRSSX99] F. Van Raamsdonk, P. Severi, M. H. Sørensen, and H. Xi. Perpetual reductions in λ-calculus. *Inform. and Comp.*, 149(2):173–229, 1999.

[WH87] P. Wadler and R.J.M. Hughes. Projections for strictness analysis. In *Proc. Functional Programming Languages and Computer Architecture*, pages 385–407. Springer-Verlag, 1987. Lecture Notes in Computer Science, Vol. 274.

[Wol93] D. Wolfram. *The Causal Theory of Types*. Cambridge Univ. Press, Cambridge, UK, 1993. Cambridge Tracts in Theoretical Computer Science, Vol. 21.

[Won00] L. Wong. The functional guts of the kleisli query system. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming*, pages 1–10. ACM Press, 2000.

[Xi96] H. Xi. An induction measure on λ-terms and its applications. Technical Report 96–192, Department of Mathematical Sciences, Carnegie Mellon University, 1996.

[Xi97] H. Xi. Weak and strong beta normalisations in typed λ-calculi. In P. De Groote and J. R. Hindley, editors, *Typed Lambda Calculus and Applications*, pages 390–404. Springer-Verlag, 1997. Lecture Notes in Computer Science, Vol. 1210.

[XP98] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'98*, pages 249–257. ACM Press, 1998.

[XP99] H. Xi and F. Pfenning. Dependent types in practical programming (extended abstract). In *Proc. 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'99*, pages 214–226. ACM Press, 1999.

[YHI93] K. Yi and W.L. Harrison III. Automatic generation and management of interprocedural program analyses. In *Proc. 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'93*, pages 246–259. ACM Press, 1993.

[Zan95] H. Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24(1–2):89–105, 1995.

[篠00] 篠埜功、胡振江、武市正人. グラフの探索関数の再帰的定義と変換. **コンピュータソフトウェア**, 17(3):2–19, 2000.

[小86] 小野諭、高橋直久. 再帰関数系における依存属性集合の計算法. **電子情報通信学会論文誌**, J69-D(5):714–723, 1986.

[小 87] 小野諭、高橋直久. 関数型言語における要求駆動型評価の並列処理向け最適化. **電子情報通信学会論文誌**, J70-D(2):259–268, 1987.

[小 88] 小川瑞史、小野諭. 広域データフロー解析に基づく関数型プログラムの変則性検出. **電子情報通信学会論文誌**, D71(10):1949–1958, 1988. English translation: Systems and Computers in Japan, Vol.21, No.2, pp.11-22, 1990.

[小 91] 小川瑞史、小野諭. 最小不動点計算に基づくプログラムの帰納的性質の導出. **情報処理学会論文誌**, 32(7):914–923, 1991.

[小 96] 小川瑞史、小野諭. 抽象実行 ーそのフレームワークと実例ー その１～その３. **コンピュータソフトウェア**, 13(2,4,6 掲載予定), 1996.

Publication Lists (2001–1987)

Tutorial Papers

1. 小川瑞史, 小野諭. 抽象実行 そのフレームワークと実例 (その１～３). コンピュータソフトウェア Vol.13, No.2, pp.3-18, No.4, pp.3-22, No.6, pp.3-25, 1996.

Journal Papers

1. Mizuhito Ogawa. Well-Quasi-Orders and Regular ω-languages, to appear in *Theoretical Computer Science*, 2002.

2. Isao Sasano, Zhenjiang Hu, Masato Takeichi, Mizuhito Ogawa. Derivation of Linear Time Algorithm for Mining Optimized Gain Association Rules. コンピュータソフトウェア掲載予定, 2002.

3. 篠埜 功, 胡 振江, 武市 正人, 小川 瑞史. 最大重み和問題の線形時間アルゴリズムの導出. コンピュータソフトウェア, Vol.18, No.5, pp.1-17, 2001.

4. 篠埜 功, 胡 振江, 武市 正人, 小川 瑞史. ナップサック問題およびその発展問題の統一的解法. コンピュータソフトウェア, Vol.18, No.2, pp.59-63, 2001.

5. Ken Mano and Mizuhito Ogawa. Unique normal form property of compatible term rewriting systems - A new proof of Chew's theorem -. *Theoretical Computer Science*, Vol.258, No.1-2, pp.169-208, 2001.

6. Zurab Khasidashvili, Mizuhito Ogawa, and Vincent van Oostrom. Perpetuality and Uniform Normalization in Orthogonal Rewrite Systems. *Information and Computation*, Vol.164, No.1, pp.118-151, 2001.

7. 松浦邦博, 大山口通夫, 太田義勝, 小川瑞史. 非線形 TRS の E 重なり性について. 電子通信情報学会論文誌 Vol.80-D-I, No.11, pp.847-855, 1997.

8. 真野健, 小川瑞史. 高階書き換え系の単一正規形性. 電子通信情報学会論文誌 Vol.80-D-I, No.3, pp.258-268, 1997.

9. 小川瑞史, 小野諭. 最小不動点計算に基づくプログラムの帰納的性質の導出. 情報処理学会論文誌 Vol.32, No.7, pp.914-923, 1991.

10. Mizuhito Ogawa and Satoshi Ono, Transformation of strictness-related analyses based on abstract interpretation. IEICE Transaction, Vol.E 74, No.2, pp.406-416, 1991.

11. 小川瑞史, 小野諭. 広域データフロー解析に基づく関数型プログラムの変則性検出. 電子情報通信学会論文誌 Vol.D 71, No.10, pp.1949-1958, 1988. (English translation: Systems and Computers in Japan, Vol.21, No.2, pp.11-22, 1990)

Conference Papers

1. Mizuhito Ogawa. Abstract interpretation over infinite abstract domains, Proceedings of the 2nd Asian Workshop on Programming Languages and Systems, pp.183-191. KAIST, Daejeon, Korea, December, 2001. ROPAS Techenical Memorandum 2001-16.

2. Mizuhito Ogawa. Linear time algorithm generation based on well-quasi-orders, - An example of query processing -, Proceedings of the Fourth International Symposium on Theoretical Aspects of Computer Software (TACS2001), Lecture Notes in Computer Science 2215, pp.283-297. Tohoku University, Sendai, October, 2001. Springer-Verlag.

3. Mizuhito Ogawa. Complete axiomatization for an algebraic construction of graphs. Proceedings of The RIEC International Symposium on Rewriting in Proof and Computation (RPC'01), pp.199-207. Itsutsubashi Kaikan, Sendai, October, 2001. RIEC report.

4. Zurab Khasidashvili, Mizuhito Ogawa, and Vincent van Oostrom. Uniform Normalization beyond Orthogonality, Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA2001), Lecture Notes in Computer Science 2051, pp.122-136. Utrecht, The Netherlands, May, 2001. Springer-Verlag.

5. Isao Sasano, Zhenjiang Hu, Masato Takeichi, Mizuhito Ogawa. Make It Practical: A Generic Linear-Time Algorithms for Solving Maximum-Weightsum Problems. Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00), pp.137-149, Montreal, September, 2000. ACM Press.

6. Mizuhito Ogawa. Well-Quasi-Orders and Regular ω-languages. Proceedings of the Third International Colloquium on Words, Languages and Combinatorics, pp.105-106, Kyoto, March, 2000.

7. Mizuhito Ogawa. Automatic verification based on abstract interpretation. Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99), Lecture Notes in Computer Science 1722, pp.131-146. Tsukuba, November, 1999. Springer-Verlag.

8. Zurab Khasidashvili and Mizuhito Ogawa. Perpetuality and Uniform Normalization. Proceedings of the 6th International Conference on Algebraic and Logic Programming (ALP'97), Lecture Notes in Computer Science 1298, pp.240-255. Southampton (England), September, 1997. Springer-Verlag.

9. Ken Mano and Mizuhito Ogawa. Unique Normal Form Property of Higher-Order Rewriting Systems. Proceedings of the 5th International Conference on Algebraic and Logic Programming (ALP'96), Lecture Notes in Computer Science 1139, pp.269-283. Aachen (Germany), September, 1996. Springer-Verlag.

10. Mizuhito Ogawa. Chew's theorem revisited - uniquely normalizing property of nonlinear term rewriting systems. Proceedings of the 1st International symposium on algorithms and computation (ISAAC'92), Lecture Notes in Computer Science 620, pp.309-318. Nagoya, December, 1992. Springer-Verlag.

11. Mizuhito Ogawa and Satoshi Ono. Transformation of strictness-related analyses based on abstract interpretation. Proceedings of the International Conference on Fifth Generation Computer Systems 1988 (FGCS'88), pp.430-438. Tokyo, December, 1988.

Domestic Conference/Workshop Papers/Manuscript

1. 小川瑞史. 非線型なメンバーシップ条件付項書換系の NVNF-逐次性. 日本ソフトウェア科学会第 18 回大会, 4A-2, September 2001.

2. Isao Sasano, Zhenjiang Hu, Masato Takeichi, Mizuhito Ogawa. Derivation of Linear Time Algorithm for Mining Optimized Gain Association Rules. 日本ソフトウェア科学会第 18 回大会, 1D-1, September 2001.

3. 篠埜 功, 胡 振江, 武市 正人, 小川 瑞史. ナップサック問題およびその発展問題の統一的解法. 日本ソフトウェア科学会第 17 回大会, C5-4, September 2000.

4. Mizuhito Ogawa. Automatic verification based on abstract interpretation. *IPSJ SIG notes, 99-PRO-24-19*, June, 1999.

5. Zurab Khasidashvili and Mizuhito Ogawa. Perpetuality and Uniform Normalization. *IPSJ SIG notes, 97-PRO-13-2*, May, 1997.

6. 小川瑞史. 逐次性 v.s. ストリクト性 -非線型項書換え系の最適戦略にむけて-. *IPSJ SIG notes, 96-PRO-10-12*, November, 1996.

7. 小川瑞史, 小野諭. 抽象解釈における Lazy な抽象領域の生成. *IPSJ SIG notes, 96-PRO-7-6*, May, 1996.

8. 真野健, 小川瑞史. 非線型高階書き換え系の単一正規形性について. *IPSJ SIG notes, 95-PRO-4-2*, November, 1995.

9. Mizuhito Ogawa. Simple gap termination for term graph rewriting systems. RIMS Workshop on Theory of Rewriting Systems and Its Applications, *Research report 918, RIMS, pp.99-108*, July, 1995. Kyoto Univ.

10. Ken Mano and Mizuhito Ogawa. A new proof of Chew's theorem. RIMS Workshop on Theory of Rewriting Systems and Its Applications, *Research report 918, RIMS, pp.160-177*, July, 1995. Kyoto Univ.

11. Mizuhito Ogawa. Simple gap termination for term graph rewriting systems. *ISPJ SIG notes, 95-PRO-1-4*, June, 1995.

12. Ken Mano and Mizuhito Ogawa. A new proof of Chew's theorem. *IPSJ SIG notes, 94-PRG-19-7*, October, 1994.

13. Mizuhito Ogawa. Simple termination with gap-condition *IPSJ SIG notes, 94-PRG-19-2*, October, 1994.

14. Mizuhito Ogawa. Polynomial-time algorithm generation based on graph minors. *IPSJ SIG notes, 93-PRG-12-9*, May, 1993.

15. 小川瑞史. 証明正規化に基づくチャーチロッサー性に関連する性質の証明. 電子情報通信学会春季大会 SD-1-4, 分冊 6, pp.263-264, Febuary, 1992.

16. 小川瑞史, 小野諭. 抽象的解釈に基づく広域解析の設計法. 日本ソフトウェア科学会シンポジウム・関数プログラミング, January, 1992.

17. Mizuhito Ogawa, Satoshi Ono. Detecting non-monotonic properties in functional programs. *IEICE SIG notes, COMP-89-107*, Febuary, 1990.

18. Mizuhito Ogawa and Satoshi Ono. On the completion algorithm for possibly non-terminating and nonlinear term rewriting systems. *IPSJ SIG notes, 89-SF-31-10*, September, 1989.

19. Satoshi Ono, Mizuhito Ogawa, and Yukio Tsuruoka. Computation Path Analysis with Path Valid Condition. *IEICE SIG notes, SS-89-167*, September, 1989.

20. Mizuhito Ogawa and Satoshi Ono. On the Uniquely Converging Property of Non-linear Term Rewriting Systems. *IEICE SIG notes, COMP-89-7*, May, 1989.

21. Mizuhito Ogawa and Satoshi Ono. Transformation of strictness-related analyses based on abstract interpretation. *IEICE SIG notes, COMP-88-16*, June, 1988.

22. 小川瑞史, 小野諭. 広域データフロー解析に基づく関数型プログラムの変則性検出. 電子情報通信学会データフローワークショップ 1987 予稿集, pp.33-40, October, 1987.

23. 小川瑞史, 小野諭. 関数型プログラムの拡張した抽象的解釈とその広域解析技術への応用. 理研シンポジウム・関数プログラミング, pp.28-37, Febuary, 1987.