Perfect Supercompilation

Master's Thesis

JENS PETER SECHER

Department of Computer Science
University of Copenhagen

Preface

This report constitutes a Master's Thesis forming part of the credit towards the Master's degree at DIKU, the Department of Computer Science at the University of Copenhagen, Denmark. The advisor on this work was Morten Heine B. Sørensen, Assistant Professor at DIKU.

This thesis is in English because I would like to believe that it is of interest to Computer Scientists in general, and Programming Language Researchers in particular. As English is not my native tongue, I hope that the reader will forgive the many mistakes I have made.

I present a technique that transforms programs by means of propagating negative as well as positive information. Some implementations of supercompilers that use ideas similar to those presented in this thesis exist, but the theory behind these transformers have never been described and these techniques are not guaranteed to terminate. My contribution is thus a very detailed description of a particular – but yet generic – approach that is guaranteed to terminate. I do not provide an implementation, and therefore there are no empirical results to show the feasibility of this theoretical approach to program transformation. I hope, however, to be able to present experimental results in the near future.

Acknowledgement

I would first of all thank my advisor for spending a tremendous amount of time on discussions and for being an example of almost superhuman discipline. I want to thank Robert Glück for discussions about process trees and restrictions, and for his inspiring articles in general. I also want to thank Laura Lafave and Michael Leuschel for inspiring discussions about program transformation and termination. I am very grateful to Neil D. Jones for his insights and perspectives, for having introduced me to a number of the best researchers in the field and for providing an excellent environment for programming-language research at DIKU. Finally, I would like to thank my daughter and her mother for their understanding during my work on this text.

Contents

Chapter 1

Introduction

From a mathematical point of view, a computer program should act as a function: a program should, when given some input value, give us an answer. This is indeed how most people would think of computation. For instance, when one presses the "recalculate" button in a spreadsheet, one would expect to see the new budget. But as everyone knows, in a large spreadsheet it might "take a while" before the result appears, and we also know that some spreadsheet programs are faster than others.

In this thesis we are concerned with making programs faster in an automatic way. To see how this can be achieved, we need to take a closer look on the process of computation itself.

Sequential execution of a program q for some input value v will proceed in a number of steps. In each step, intermediate results can be calculated and saved for later use by modifying the computer's *store* (*i.e.* the *state* of the computer). If we let s_0 represent the initial store, we can picture the computation as a sequence of states and transitions

$$\text{execute}(q, v) : s_0 \to s_1 \to s_2 \to \ldots \to s_n$$

The result of the computation can read off from the last store s_n. Of course, if the computation process never terminates, there will be no last store and we will not be able to get a result.

If we, say, by inspection of the program alone, are able to predict that there is a pattern in some subsequence $s_m \to s_{m+1} \to \ldots \to s_l$ that will be repeated several times *regardless of the actual input*, we can speed up *any* such computation by *altering* q so that it performs such sub-computation once and for all. The result of this *optimisation* (usually called *common-subexpression elimination*) can be depicted as replacing all but the first of these repeated patterns by the result of the first pattern, thus shortening the computation sequence. In everyday terms, this is somewhat analogous to storing intermediate calculations on a pocket calculator.

When we take also the *data* used or produced by the computation process into account, more advanced optimisation techniques can be employed. Consider two program fragments f and g where f produces a result that is solely used by g.

$$\text{data}_1 \xrightarrow{\quad f \quad} \text{data}_2 \xrightarrow{\quad g \quad} \text{data}_3$$

The allocation (and deallocation) of the intermediate data is unnecessary and time consuming. By a technique called *deforestation*, traversal of such intermediate data can be eliminated by replacing two sequential computation processes by their composition:

$$\text{data}_1 \xrightarrow{\quad g \circ f \quad} \text{data}_3$$

A related, but orthogonal, problem is parallel traversal of data:

$$\text{data}_1 \xrightarrow{\ f\ } \text{data}_2$$

$$\text{data}_1 \xrightarrow{\ g\ } \text{data}_3$$

A technique called *tupling* eliminates such repeated traversals by only traversing the data once:

$$\text{data}_1 \xrightarrow{\ \langle f,g\rangle\ } \langle \text{data}_2, \text{data}_3\rangle$$

In this text we will present a technique called *supercompilation* that encompasses deforestation and to some extend achieves optimisations similar to tupling. The aim of this endeavour is to automatically speed up the computation process.

1.1 Supervised Compilation

The idea of automatically transforming programs, *i.e.* treating programs as data objects, appeared even before modern computers existed, and the idea has resurfaced again in different shapes and flavours ever since. In this section we will review one such flavour and try to explain the philosophy undermining this school.

The Supervised Compilation Project was started in the 1960s by V. F. Turchin as an approach to *artificial intelligence* [33]. Turchin[1] divided creative thinking, the cornerstone of intelligence, into three parts:

1. Deduction.
2. Induction.
3. Meta-system Transition.

The explanation for this is as follows. When solving a certain problem, we will apply our knowledge (a set of rules) to the problem at hand, and use the rules to *deduce* some answer to the problem. When a pattern appears, we might *induce* new rules that are more direct than the general rules we started out with, thus accumulating new knowledge. If we cannot find a solution to the problem at hand, we might take a step back and analyse why we failed: there could be something wrong with *the way* we use our rules; maybe new *ways* of applying our rules should be invented. Such a step is called a *Meta-system Transition* because we leap from reasoning about normal objects to considering the rules themselves as the objects of interest. In effect, we introduce meta-rules, *i.e.* rules about the rules. If we do not succeed this way, we may make yet another meta-system transition, finding new ways to introduce meta-rules. This process could in principal go on indefinitely, for instance, if the problem cannot be solved.

Supervised Compilation (supercompilation, for short) aimed at implementing the above ideas on a computer: deduction can be carried out by clever computation (called driving), induction can be simulated by generalisation, and meta-system transition can be achieved by writing interpreters for metalanguages, *i.e.* more abstract languages. The fundamental language used

[1]The following interpretation of Turchin's work is very simplified and is most likely not how he would describe his own work.

by Turchin, *i.e.* the predefined language, is called Refal. This language is a (somewhat strange) rewrite system on strings [29].

The Refal language could thus be used as a *specification language* for programming languages: An interpreter for a language L could be written in Refal, and supercompilation could turn L-programs into efficient programs. In fact, one could imagine to "introduce a hierarchy of ad hoc programming languages specialized for the current problem" [32], by letting language L_n be defined as an interpreter written in language L_{n-1}. Furthermore, since the Refal supercompiler is self-applicable, Turchin independently discovered was has become known as the Futumura projections [9].

In this thesis we will stay on the ground, so to speak, and concentrate on what we believe to be the core of supercompilation, namely the program transformation techniques that involve driving and generalisation. We will present an automatic, sophisticated supercompiler algorithm that propagates negative as well as positive information. Moreover, the algorithm is guaranteed to terminate.

1.2 Applications

It has been shown [35, 24, 11, 27] that by supercompilation one can achieve the effects of various other program transformation techniques, *e.g.*

- Elimination of intermediate data structures [37],
- Program specialization [13],
- Theorem Proving [31]

and we will see examples of some of these effects in the following sections.

In terms of software engineering, program transformation could prove to be a novel tool, because it can be used to ensure the efficiency of highly specialized or abstract programming languages. If the programmer knows that the machine will take care of all the messy details – and do it well – he can focus on creating programs that are easy-to-understand and highly maintainable. One way to accomplish this is by implementing small interpreters for domain specific languages, which can then be used as fourth-generation languages. This idea is not new, but by supercompilation we can ensure that the programs written in such metalanguages are efficient by means of *specialization*, as shown in [34].

Program transformation attracts increasing attention in industri, as can be seen *e.g.* in the Intentional Programming Group at Microsoft Research. In this group, unfold/fold transformation techniques are employed to ensure efficiency when programming at a high level of abstraction. From this work [23] it is clear that, for a program transformer to be successful, two conditions must be fulfilled: It must be automatic and it must be powerful.

We will present a such a program transformer: It is automatic since it is guaranteed to terminate, and it powerful because it propagates perfect information. The perfectness lies in that, in addition to positive information, also negative information is propagated. Recently, Augustsson has reported that negative information propagation is essential in an Aircraft Crew Planning system [2], and we will see that negative information propagation removes redundant tests in programs. To our knowledge, no other supercompiler that propagates negative information has been described in full or guaranteed to terminate.

1.3 Overview

In chapter two, we will give an introduction to supercompilation and give a tast of the merits of this transformation technique and the problems that will be encountered.

In chapter three, our object language, a small functional language, is presented and semantics discussed.

Chapter four will give an in-depth treatment of *restriction systems*, which are aggregates used to collect information during driving. We present a rewrite system that normalises such restriction systems, and we show that it can be decided whether the constraints in a restriction system can be satisfied.

In chapter five, we will use restriction systems and the semantics of the language to define the core machinery of the supercompiler, the *driving* algorithm. We show that this driving mechanism can produce an optimal KMP pattern matcher from a naïve pattern matcher.

Chapter six will present an algorithm ensures termination of the supercompiler by means of generalisations.

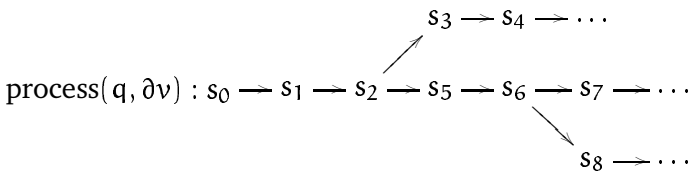
In chapter seven, we prove that the algorithm indeed terminates by considering abstract program transformers in the metric space of trees.

Chapter eight explains how the transformed program will be produced, and finally in chapter nine we conclude and refer to related work.

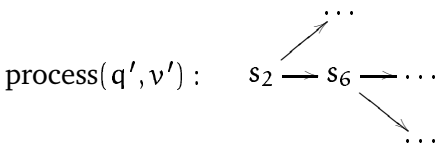
Chapter 2

Program Transformation

With the computational model presented in the introduction, we can describe the execution of some program q with input v as sequence of computation steps. Our supercompiler will, however, *simulate* the execution of a program on the basis of partial knowledge about the input. This means that we no longer can view computation strictly as a sequence, but we must instead consider non-deterministic computation because some decisions cannot be made. If we let ∂v denote imperfect knowledge about the input, the simulated execution of q can be modelled as a tree of computations:



From this *tree of states* [32] we can extract a new program that behaves like the original program (with regard to the partial input), but does not perform the intermediate steps:



Even in the the seemingly uninteresting case where we have no knowledge about the input, we might gain a reduction in the total number of steps performed by first simulating the execution, and then from this simulation extract a new program.

Programs transformed in this manner are not very readable, but this is usually of no consequence since they will be *semantically equivalent* to the original program with respect to the partial knowledge about the input. Program transformation can thus be compared to the output from a standard compiler: except for debugging purposes, no-one is interested in *reading* the output from the compiler, as long as the *execution* of the object code does what it is supposed to.

In this thesis we shall be concerned with a small functional language, so the above notion of a computer store can be replaced by the more abstract notion of terms in a *term rewrite system* (see *e.g.* [15]).

2.1 Example of Supercompilation

We will explain supercompilation in the accordance with the general framework for on-line program transformation presented by Burstall and Darlington [3][1]. This ground-breaking work has been the inspiration for a number of other frameworks, of which the best suited for our purpose is the one presented by Pettorossi and Proietti [21].

Following Pettorossi and Proietti, we can view the transformation process as consisting of three conceptual phases: *symbolic computation*, *search for regularities,* and *program extraction*.

In the first phase, a possibly infinite *process tree* is constructed such that it depicts all possible execution traces of the program at hand. Each node in the tree is produced by *unfolding* the parent node. In the second phase, the tree is pruned by means of *generalisation* to ensure that the process tree is finite. In the third phase, a new program is *extracted* from the finite tree.

We will now give examples of supercompilation on functional programs in this framwork. The first two examples are taken almost verbatim from [26].

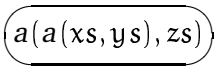
Example 1
Consider a program for appending two lists:

$$\begin{array}{lcl} a([\,],vs) & = & vs \\ a(u{:}us,vs) & = & u{:}a(us,vs) \end{array}$$

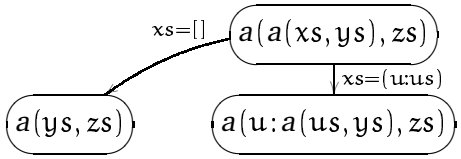
where $[\,]$ is the notation for an empty list, and $x : y$ is the notation for a list with head x and tail y.

If we want to append three lists, it can be achieved by the term $a(a(xs,ys),zs)$. This would, however, be inefficient, since the list xs would be traversed twice. We will now illustrate how we can transform this term into a more efficient one by supercompilation.

We start out with a process tree that contains a single node, namely one that is labelled with the term we want to transform.



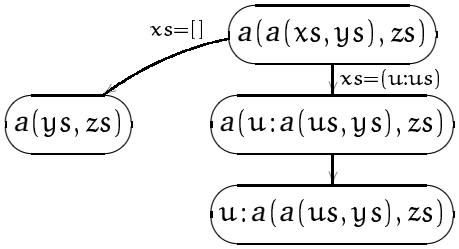
By *unfolding* the inner call according to the definition of a, we obtain two different terms, corresponding to the possible outcomes of the application. We now create two nodes labelled by the results of the unfold operation, and add these nodes as children to the node that was unfolded:



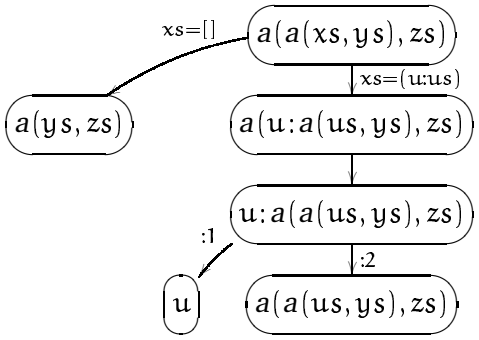
Each edge is labelled by the exact operation that produced the child, in this case an instantiation of the variable xs. In the rightmost child we can perform another unfolding step on the

[1] In the Burstall & Darlington framework, the subject program is modified incrementally during the transformation. This corresponds somewhat to the ideas presented by Turchin, where accumulated knowledge about programs are stored as models of the programs.

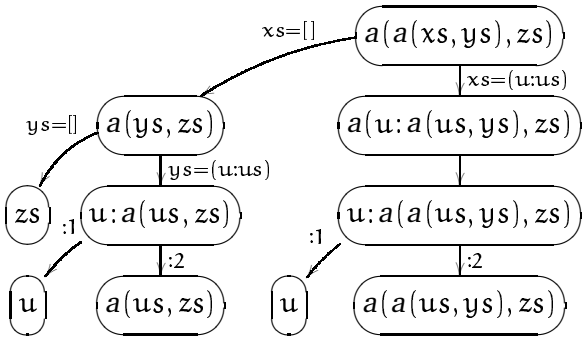
outer call to append:



The label of the new child contains an outermost constructor. For transformation to propagate to the subterm of the constructor we again add children:

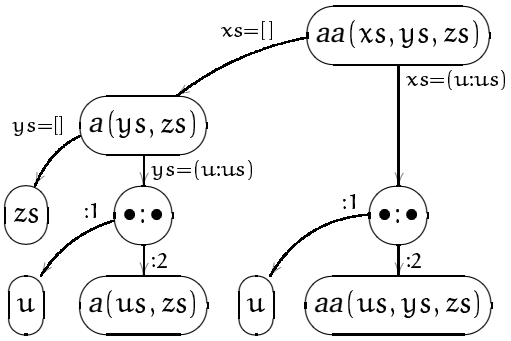


The labels on the new edges reflect that it was the constructor ":" that produced new children. The term in the rightmost child is a renaming of the term in the root; that is, the two terms are identical up to choice of variable names. As we shall see below, no further processing of such a node is required. Unfolding the child with label $a(ys, zs)$ two steps leads to:



The process tree is now *closed* in the sense that each leaf term either is a renaming of an ancestor's term, or contains a variable or a 0-ary constructor, which cannot be further unfolded. Informally, a closed process tree is a representation of all possible computations with the term t in the root, where branches in the tree correspond to different run-time values for the free variables of t.

To construct a new program from a closed process tree, we introduce for each branching node δ with child γ a definition where the left and right hand side of the definition are derived from δ and γ, respectively. More specifically, in the above example we rename terms of form $a(a(xs, ys), zs)$ as $aa(xs, ys, zs)$, and derive the following process tree:



From the labels on the edges we can now construct the following program:

$$
\begin{array}{lcl}
aa([\,], ys, zs) & = & a(ys, zs) \\
aa(u{:}us, ys, zs) & = & u : aa(us, ys, zs)
\end{array}
$$

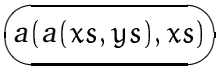
$$
\begin{array}{lcl}
a([\,], zs) & = & zs \\
a(u{:}us, zs) & = & u{:}a(us, zs)
\end{array}
$$

The term $aa(xs, ys, zs)$ in this program is more efficient than $a(a(xs, ys), zs)$ in the original program, since the new term traverses xs only once. We have thus eliminated the creation of the intermediate list resulting from the term $a(xs, ys)$ because the transformation discovered that this list is consumed by the outer call to append. Such elimination of intermediate data structures is collectively called *deforestation* [37].

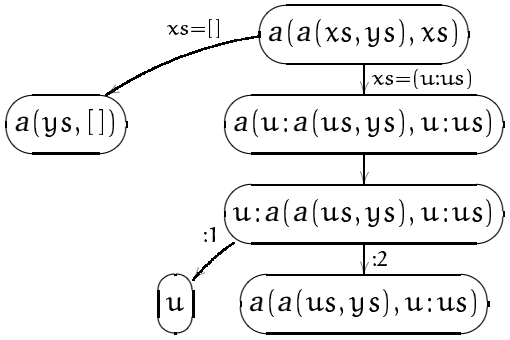
The transformation in Example 1 proceeded in three phases – symbolic computation, search for regularities, and program extraction – where the first two were interleaved. In the first phase we performed unfolding steps that added children to the tree. In the second phase we made sure not to unfold a node with a term which was a renaming of ancestor's term, and we continued the overall process until the tree was closed. In the third phase we recovered from the resulting finite, closed tree a new term and program.

In the above transformation we ended up with a finite closed process tree. Often, special measures must be taken to ensure that this situation is eventually encountered.

Example 2
Suppose we want to transform the slightly different term $a(a(\mathbf{xs}, ys), \mathbf{xs})$, where a is defined as in Example 1. As above we start out with:

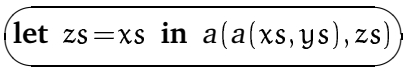


After the first few steps we have:

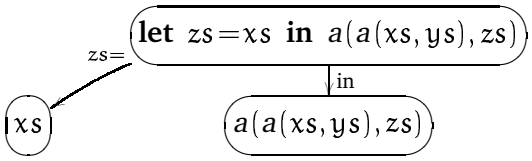


Unlike the situation in Example 1, the label of the rightmost node is not a renaming of the term at the root. Repeated unfolding the right-most branch would, in fact, *never* lead to a situation where the leaf is a renaming of some ancestor, and the supercompilation would never terminate.

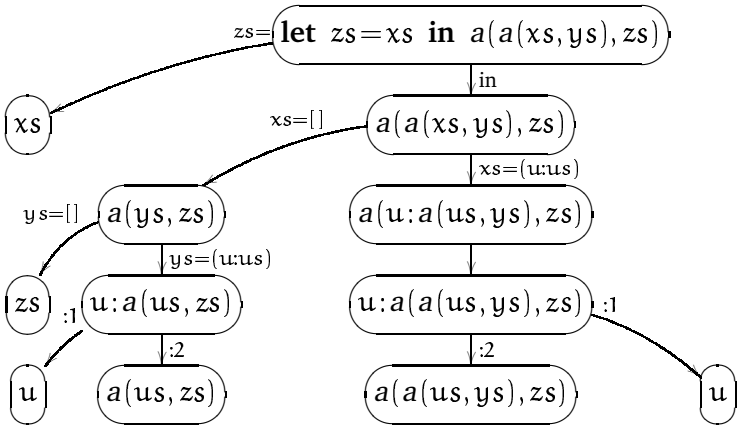
One solution to this problem is to ignore the information that the argument xs to the inner call and the argument xs to the outer call are the same. This is achieved by a *generalisation* step that replaces the whole process tree by a single new node:



When dealing with nodes of the new form **let** $zs = t$ **in** t' we then transform t and t' independently. Thus we arrive at:



Unfolding of the node labelled $a(a(xs,ys),zs)$ leads to the same process tree as in Example 1:



When generating a new term and program from such a process tree, we could eliminate all let-terms by substituting every let-bound variable with the appropriate term. This could, however, lead to duplication of terms, a subject we will get back to in chapter 8. To play it safe, we will instead introduce a new function definition for each let-term. From the above tree we can then generate the term $f(xs,xs,ys)$ and the following program:

$$
\begin{array}{lcl}
f(zs,xs,ys) & = & aa(xs,ys,zs) \\
 & & \\
aa([\,],ys,zs) & = & a(ys,zs) \\
aa(u{:}us,ys,zs) & = & u : aa(us,ys,zs) \\
 & & \\
a([\,],zs) & = & zs \\
a(u{:}us,zs) & = & u{:}a(us,zs)
\end{array}
$$

This is essentially the same program as in Example 1.

Again transformation proceeds in three phases, but the second phase is now more sophisticated, sometimes replacing a subtree by a new node due to a generalisation step.

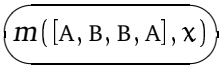
In these two examples, an unfolding step resulted in instantiation of variables, which made it possible to limit the number of choices in a subsequent unfolding step. In fact, the limitation of choices was the reason that the transformed program was more efficient than the original, since some unfoldings could be carried out regardless of the actual arguments. We will now see how the number of choices can be limited by more subtle information.

Example 3
Let us adopt the convention that $[x, y, z]$ denotes the lists $x : (y : (z : []))$, and assume that we have a set $\{\text{TRUE}, \text{FALSE}, \text{A}, \text{B}, \text{C}, \ldots\}$ of values at our disposal.

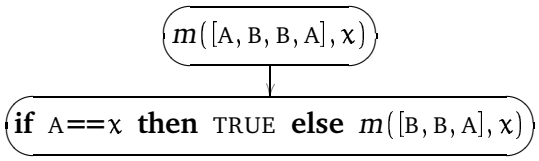
Consider now a program that tests for membership in a lists:

$$\begin{array}{lcl} m([\,], v) & = & \text{FALSE} \\ m(u{:}us, v) & = & \textbf{if } u{=}{=}v \textbf{ then } \text{TRUE} \textbf{ else } m(us, v) \end{array}$$

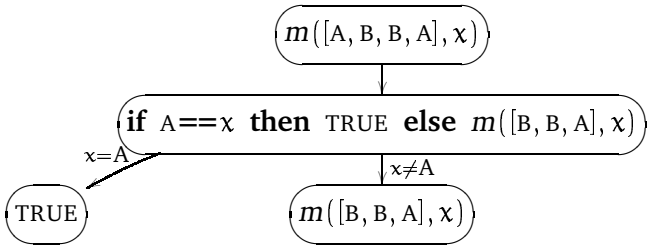
If we would like test for membership in the list $[\text{A}, \text{B}, \text{B}, \text{A}]$ it can be done by the term $m([\text{A}, \text{B}, \text{B}, \text{A}], x)$. Transformation of this term will again start with the a single node:



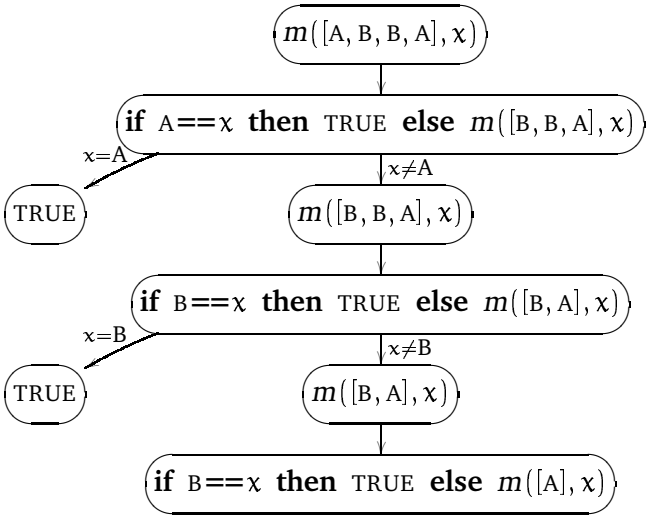
An unfolding step will only add a single child because the first argument is different from the empty list.



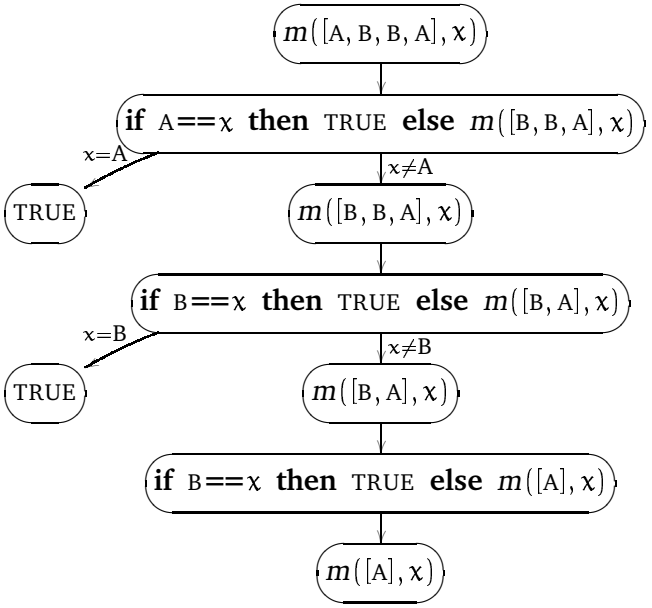
Since the condition cannot be decided upon, we will add two children to reflect the possible outcomes of the test.



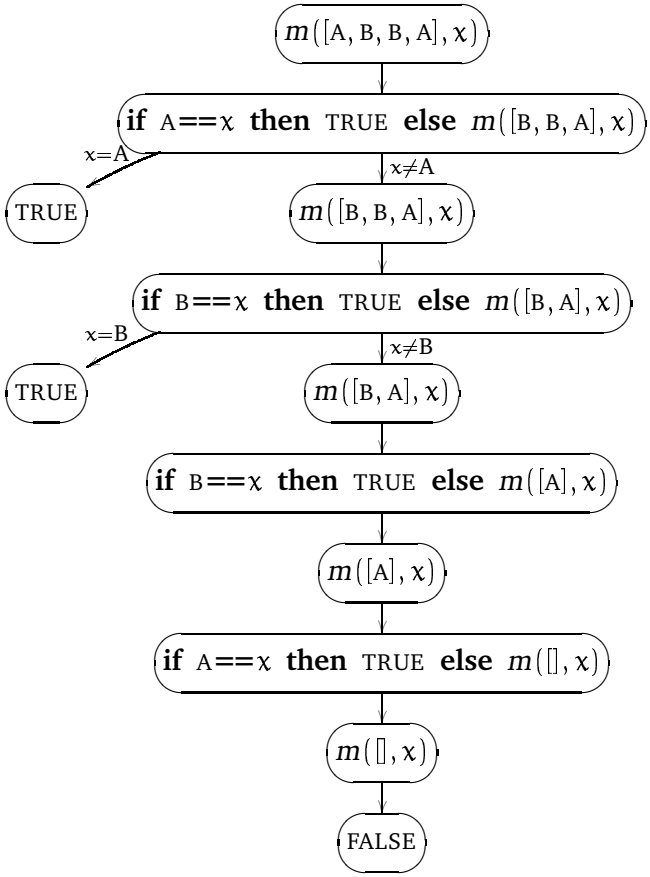
Each edge is again labelled by the operation that produced the child, but notice how the *negative* information $x \neq \text{A}$ for the right child cannot be represented by an instantiation. We continue the development of the process tree:



We are now about to unfold the leaf containing a conditional, and the situation seems to be similar to what we have previous encountered since we do not know the value of x. However, from the node right above the leaf we are about to unfold, we know that $x \neq B$. We can thus add a single child to the process tree.



As we now proceed, we can utilise the information that also $x \neq A$. The process tree will have following development:



The process tree is now closed, and we can construct a new program by inspection of the labels on the edges:

$$mabba(v) \quad = \quad \textbf{if } \text{A}{==}v \textbf{ then } \text{TRUE } \textbf{else if } \text{B}{==}v \textbf{ then } \text{TRUE } \textbf{else } \text{FALSE}$$

This program is more efficient than the original since the traversal of the list is eliminated.

Again transformation proceeds in three phases, but we utilised *negative* information ($x \neq \text{A}$ and $x \neq \text{B}$) to avoid unnecessary test in the transformed program. Unlike *positive* information (*e.g.* $x = \text{A}$), negative information cannot be propagated to a term by means of instantiation. We will therefore need some aggregate that can collect negative information. In the chapter 4, we will present such an aggregate.

Chapter 3

Object Language

One of the reasons that supercompilation did not attract much attention until 1995, is undoubtedly that was based on the (for many) unfamiliar language Refal. Recent work has been conducted to remove this hindrance [24, 27, 26], and we shall continue in this spirit. Besides recasting the ideas in a (for most people) familiar setting, we feel that it is necessary to treat only the very essential features of a programming language, so that the description of the program transformation technique avoids unnecessary cluttering of the basic concepts – these can be complicated enough as it is.

We shall therefore be concerned with a simple language which is a first-order functional language with a predefined comparison operation. There are no predefined values, but type definitions can introduce new sets of values in an obvious way.

The limitation of treating only a tiny language has both strengths and weaknesses. The simplicity of the object language makes it easier to show theoretical results because we are liberated from language specific problems and peculiarities. On the other hand, the simplicity of the object language makes it cumbersome to produce good examples because the expressiveness is rather limited.

3.1 Syntax

Definition 4
The syntax of our language is defined in figure 3.1, which should be read together with the following definitions and restrictions on the language:

1. A constructor $\mathtt{c}$ must only appear in one type definition, and must only be used once in this type definition.

2. A function name $\mathtt{f}$ must only be used for one function or pattern function definition.

3. All variables on the left-hand side of a function definition must be distinct.

4. All variables on the right-hand side of a function definition must appear on the left-hand side. Likewise for type definitions.

5. The patterns in pattern functions must be disjoint, meaning that a particular constructor can appear in at most one pattern.

6. A program must be type correct with regard to the type system defined in section 3.2.

Let $x \in \mathscr{X}$ range over variables, $f \in \mathscr{F}$ range over function names, $\psi \in \Psi$ range over type names, $c \in \mathscr{C}$ range over constructors, $\alpha \in \mathscr{A}$ range over type variables.

(programs)	$\mathscr{Q}$	$\ni$	q	$::=$	$S_1 \ldots S_n\ d_1 \ldots d_m$	
(type definitions)	$\mathscr{S}$	$\ni$	S	$::=$	**data** $\psi(\alpha_1, \ldots, \alpha_n) = K_1 \mid \ldots \mid K_m$	
(constructors)	$\mathscr{K}$	$\ni$	K	$::=$	$c(\tau_1, \ldots, \tau_n)$	
(types)	$\mathscr{T}$	$\ni$	τ	$::=$	α	(type var)
				$\mid$	$\psi(\alpha_1, \ldots, \alpha_n)$	(usertype)
(definitions)	$\mathscr{D}$	$\ni$	d	$::=$	$f(x_1, \ldots, x_n) = t$	(function)
				$\mid$	$f(p_1, x_1, \ldots, x_n) = t_1$	
					$\vdots$	(pattern funct.)
					$f(p_m, x_1, \ldots, x_n) = t_m$	
(terms)	$\mathscr{T}$	$\ni$	t, s, u	$::=$	x	(variable)
				$\mid$	$c(t_1, \ldots, t_n)$	(construction)
				$\mid$	$f(t_1, \ldots, t_n)$	(application)
				$\mid$	**if** $t_1 == t_2$ **then** t_3 **else** t_4	(conditional)
(patterns)	$\mathscr{P}$	$\ni$	p	$::=$	$c(x_1, \ldots, x_n)$	

Figure 3.1: The source language grammar ($n \geq 0, m > 0$)

This definition means that equality on terms must be expressed in terms of a conditional construct, since function definitions like $f(x, x) = \ldots$ are illegal. Also, the definition implies that, even though several right-hand sides for the same pattern function can exist, there is no choice in the selection of which function body to evaluate since the patterns must be disjoint.

The conditional may seem strange, since we have declared that we deliberately want a very simple language. Besides the convenience that the programmer does not have to define a comparison operation on every data type, this construct makes it possible to propagate negative information, as we shall see in chapter 5.

Example 5
An implementation of a naïve string matcher could be expressed in the following way:

$$
\begin{array}{lcl}
\textbf{data}\ \mathrm{Bool}() & = & \textsc{true} \mid \textsc{false} \\
\textbf{data}\ \mathrm{Letter}() & = & \textsc{a} \mid \textsc{b} \mid \textsc{c} \mid \ldots \\
\textbf{data}\ \mathrm{List}(a) & = & \textsc{cons}(a, \mathrm{List}(a)) \mid \textsc{nil} \\
 & & \\
\mathit{match}(p, s) & = & \mathit{m1}(p, s, p, s) \\
 & & \\
\mathit{m1}(\textsc{nil}, s, op, os) & = & \textsc{true} \\
\mathit{m1}(\textsc{cons}(p, pp), s, op, os) & = & \mathit{m2}(s, p, pp, op, os) \\
 & & \\
\mathit{m2}(\textsc{nil}, p, pp, op, os) & = & \textsc{false} \\
\mathit{m2}(\textsc{cons}(s, ss), p, pp, op, os) & = & \textbf{if}\ s == p\ \textbf{then}\ \mathit{m1}(pp, ss, op, os)\ \textbf{else}\ \mathit{next}(os, op) \\
 & & \\
\mathit{next}(\textsc{cons}(s, ss), op) & = & \mathit{m1}(op, ss, op, ss)
\end{array}
$$

Given a string pattern p and a string s, the program will search through s until it either finds the pattern p, in which case TRUE is returned, or it reaches the end of s, in which case FALSE is

returned.

Notice how the pattern function *next* is only defined for non-empty lists, *i.e.* the function is only defined for one of the two possible list constructors. Also, the one-pattern-per-function restriction forces us to introduce an auxiliary function *m2*. Well-known methods [1, 36] exist for transforming arbitrary patterns into the restricted form we require. The exclusion of local definitions (*e.g.* where-clauses) can be remedied by a technique called lambda-lifting [12].

In the rest of this text we will need some standard notation to be able to treat terms in a formal manner.

Definition 6 (Equality, free variables, values, substitutions)

1. Syntactic equality on terms is denoted by the relational symbol $\equiv$.
2. Given a term t, $\mathsf{Var}(t)$ means the set of variables in t.
3. A term t is *closed* if and only if $\mathsf{Var}(t) = \emptyset$.
4. A *value* $v \in \mathscr{V}$ is a closed term that consists entirely of constructors, *i.e.* $v ::= c(v_1, \ldots, v_n)$.
5. A *substitution* θ is a total, idempotent mapping from variables to terms, where only finitely many variables x are mapped to terms different from x. Application of substitutions are written postfix, and application is extended to terms in the obvious way. Let Θ denote the set of all substitutions.
6. The support of a substitution θ is defined as $\mathsf{support}(\theta) = \{x \mid x\theta \not\equiv x\}$.
7. By $\{x_1 := t_1, \ldots, x_n := t_n\}$ we denote a substitution θ with support $\{x_1, \ldots, x_n\}$ such that $x_i\theta \equiv t_i$.
8. The *update* $\rho \oplus \theta$ of two substitutions θ and ρ is a substitution defined by

$$x(\rho \oplus \theta) = \begin{cases} x\theta & \text{if } x \in \mathsf{support}(\theta) \\ x\rho & \text{otherwise} \end{cases}$$

Example 7
Let $t = \mathrm{P}(x, y)$. Then

1. $t\{x:=z\}\{x:=y, z:=\mathrm{TRUE}\} \equiv \mathrm{P}(\mathrm{TRUE}, y)$
2. $\mathsf{Var}(t\{x:=z\}\{z:=\mathrm{TRUE}\}) = \{y\}$
3. $t(\{z:=\mathrm{TRUE}, y:=\mathrm{FALSE}\} \oplus \{x:=z\}) \equiv \mathrm{P}(z, \mathrm{FALSE})$

Given some closed term, say $\mathit{match}(\mathrm{CONS}(\mathrm{C}, \mathrm{NIL}), \mathrm{CONS}(\mathrm{A}, \mathrm{CONS}(\mathrm{C}, \mathrm{CONS}(\mathrm{B}, \mathrm{NIL}))))$ the term can be evaluated by execution of the program in example 5. The evaluation would in this case return the value TRUE.

3.2 Types

The type system defined here will allow functions and data types to be used in a *polymorphic* way, which is commonly considered one of the strengths of functional languages because it makes it possible to write generic functions. For instance, the pattern matching function in example 5 is in itself not limited to lists of letters; it can be applied to lists of any type (defined in the containing program). To allow polymorphic types, we now define the notion of a type scheme.

Definition 8

1. Arrow types and type schemes are defined by the following grammar:

$$\begin{array}{llcll} \text{(arrow type)} & \vec{\mathcal{T}} & \ni & \vec{\tau} & ::= (\tau_1, \ldots, \tau_n) \to \tau_0 \\ \text{(type schemes)} & \Sigma & \ni & \sigma & ::= \forall(\alpha_1, \ldots, \alpha_m).\vec{\tau} \\ & & & & \quad \text{where } \{\alpha_1, \ldots, \alpha_m\} = \mathsf{Var}(\vec{\tau}) \end{array}$$

   where $n, m \geq 0$.

2. An arrow type $\vec{\tau}$ is an *instance* of a type scheme $\sigma' = \forall(\alpha_1, \ldots, \alpha_n).\vec{\tau}'$, denoted $\sigma < \vec{\tau}$, if $\vec{\tau} = \vec{\tau}'\theta$ for some type substitution $\theta : \mathscr{A} \to \mathcal{T}$. Type substitutions are lifted from $\mathscr{A} \to \mathcal{T}$ to $\vec{\mathcal{T}} \to \mathcal{T}$ as usual.

3. A *type environment* E is a map from names $(\mathscr{X} \cup \mathscr{F} \cup \mathscr{C})$ to type schemes written $[x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n]$ or shortened $[x_i \mapsto \sigma_i]$.

4. A program $q = S_1 \ldots S_n \; d_1 \ldots d_m$ is *type correct* if there exists a type environment E such that $E \vdash S_i$ for all $i \leq n$ and $E \vdash d_j$ for all $j \leq m$ by the inference system defined in figure 3.2.

We will demand that a program be accompanied by a type environment that assigns so called *principal* type schemes to constructors and function names, *i.e.* the types must be as general as possible. This requirement is imposed simply to avoid introducing a full type-inference algorithm. For a discussion about algorithms for automatically deducing principal types, see [20, 6].

Example 9
If the string-matcher program in example 5 is accompanied by the (principal) type environment

$$E = \left[\begin{array}{l} \text{TRUE} \mapsto \text{Bool}(), \\ \text{FALSE} \mapsto \text{Bool}(), \\ \text{A} \mapsto \text{Letter}(), \\ \ldots \\ \text{NIL} \mapsto \text{List}(\alpha), \\ \text{CONS} \mapsto (\alpha, \text{List}(\alpha)) \to \text{List}(\alpha), \\ \mathit{match} \mapsto (\text{List}(\alpha), \text{List}(\alpha)) \to \text{Bool}(), \\ \mathit{m1} \mapsto (\text{List}(\alpha), \text{List}(\alpha), \text{List}(\alpha), \text{List}(\alpha)) \to \text{Bool}(), \\ \mathit{m2} \mapsto (\text{List}(\alpha), \alpha, \text{List}(\alpha), \text{List}(\alpha), \text{List}(\alpha)) \to \text{Bool}(), \\ \mathit{next} \mapsto (\text{List}(\alpha), \text{List}(\alpha)) \to \text{Bool}() \end{array}\right]$$

the string-matcher program will be a type correct by the inference system in figure 3.2.

We will later on need to distinguish between variables in a program based on how many different values a particular variable can assume. More specifically, we will need to divide variables in two categories: those that might assume infinitely many values, and those that definitely assume only finitely many values. This classification can be achieved by annotating variables with the types deduced by the type system. Let us therefore introduce the notion of a *finite type*.

Definition 10 (Finite type)
A *finite type* is a type that ranges over finitely many values. Every type that is not a finite type is said to be an *infinite type*.

Let $h \in (\mathscr{F} \cup \mathscr{C})$ denote both function names and constructors. $\boxed{E \vdash t : \tau}$

$$\text{(var)}\ \frac{E(x) = \tau}{E \vdash x : \tau} \qquad \text{(consfun)}\ \frac{E(h) = \sigma \qquad \sigma < \tau}{E \vdash h : \tau}$$

$$\text{(app)}\ \frac{E \vdash h : (\tau_1, \ldots, \tau_n) \to \tau \qquad \forall i \in \{1, \ldots, n\}. E \vdash t_i : \tau_i}{E \vdash h(t_1, \ldots, t_n) : \tau}$$

$$\text{(cond)}\ \frac{\forall i \in \{1,2\}. E \vdash t_i : \tau' \qquad \forall i \in \{3,4\}. E \vdash t_i : \tau}{E \vdash \textbf{if}\ t_1 == t_2\ \textbf{then}\ t_3\ \textbf{else}\ t_4 : \tau}$$

$\boxed{E \vdash S}$

$$\text{(data)}\ \frac{\forall i \in \{1, \ldots, m\}. E \vdash_{\psi(\alpha_1, \ldots, \alpha_n)} K_i}{E \vdash \textbf{data}\ \psi(\alpha_1, \ldots, \alpha_n)\ =\ K_1 \mid \ldots \mid K_m}$$

$\boxed{E \vdash_{\psi(\alpha_1, \ldots, \alpha_n)} K}$

$$\text{(consdef)}\ \frac{E(c) = \forall(\alpha_1, \ldots, \alpha_n). [(\tau_1, \ldots, \tau_m) \to \psi(\alpha_1, \ldots, \alpha_n)]}{E \vdash_{\psi(\alpha_1, \ldots, \alpha_n)} c(\tau_1, \ldots, \tau_m)}$$

$\boxed{E \vdash d}$

$$\text{(fundef)}\ \frac{E(f) = \forall(\alpha_1, \ldots, \alpha_m).(\tau_1, \ldots, \tau_n) \to \tau \qquad E[x_i \mapsto \tau_i] \vdash t : \tau}{E \vdash f(x_1, \ldots, x_n) = t}$$

$$\text{(patfundef)}\ \frac{\begin{array}{c} E(f) = \forall(\alpha_1, \ldots, \alpha_m).(\tau_0, \tau_1, \ldots, \tau_n) \to \tau \qquad p = c(y_1, \ldots, y_l) \\ E(c) = \forall(\beta_1, \ldots, \beta_{m'}).(\tau'_1, \ldots, \tau'_l) \to \tau_0 \qquad E[x_i \mapsto \tau_i][y_j \mapsto \tau'_j] \vdash t : \tau \end{array}}{E \vdash f(p, x_1, \ldots, x_n) = t}$$

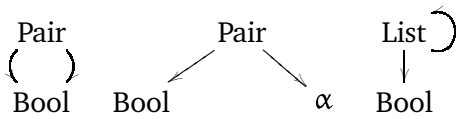
Figure 3.2: Type system for the object language. $n, m, l \geq 0$.

Informally, a finite type is characterised by its type being representable as a finite, non-cyclic graph where no leaf is labelled by a type variable.

Example 11
Consider the following type definitions:

$$\begin{array}{lcl} \textbf{data}\ \mathrm{Bool}() & = & \textsc{true} \mid \textsc{false} \\ \textbf{data}\ \mathrm{Pair}(a,b) & = & \mathrm{P}(a,b) \\ \textbf{data}\ \mathrm{List}(a) & = & \textsc{cons}(a,\mathrm{List}(a)) \mid \textsc{nil} \end{array}$$

The type Pair(Bool(),Bool()) is a finite type, whereas both Pair(Bool(),α) and List(Bool()) are infinite types. Graphically, the types can be represented thus:



An infinite type is thus called infinite because it potentially has infinitely many values.

3.3 Semantics

The intended semantics of our language is normal-order graph reduction to weak head normal form (so-called lazy evaluation). Such a semantics, however, is somewhat cumbersome to express, so we will instead present a standard normal-order reduction, which can be expressed in a very straight forward way which will suit our needs. The only difference between these two models of reduction lies in the potential duplication of computation in the latter. Consider the function definition

$$f(x) = add(x,x)$$

where the function *add* adds two numbers. If we evaluate the term f(bigcalculation) by graph reduction, we will only evaluate the time consuming calculation once, whereas the call-by-name reduction strategy presented here will evaluate it twice. In lieu of this, one should think of the semantics presented in this section as implemented by a graph-reduction. For a treatment of graph-reduction, see *e.g.* [18, 22].

Definition 12 (Evaluation)
Evaluation of a term in the object language is conducted by repeated application the $\Rightarrow_{\mathrm{cbn}}$ rule defined by the inference system in figure 3.3.

This definition of evaluation assumes that there is an outer "print loop" that outputs constructors as soon as they appear: if the term under evaluation has an outermost constructor, this constructor can be printed out and rule 11 can then be used to further evaluate the subterms until the whole term is fully evaluated. If a term does not contain an outermost constructor, it will contain a unique *redex* that will be rewritten by rules 1, 2, 4 or 5. The redex is thus either a unique function call or a conditional, which can then be unfolded. The inference rules 4–9 ensure that both operands of an equality test is evaluated to values before the comparison is carried out.

Since we assume that both programs and closed terms to be evaluated are type correct, the only way the evaluation can get stuck is by an incomplete function definition, *e.g.* as in the function *next* in example 5.

$\boxed{t \to t}$

$$(1)\ \frac{f(x_1,\ldots,x_n) \;=\; s \in q}{f(t_1,\ldots,t_n) \to s\{x_1{:=}t_1,\ldots,x_n{:=}t_n\}}$$

$$(2)\ \frac{f(c(x_1,\ldots,x_m),x_{m+1},\ldots,x_n) \;=\; s \in q}{f(c(t_1,\ldots,t_m),t_{m+1},\ldots,t_n) \to s\{x_1{:=}t_1,\ldots,x_n{:=}t_n\}}$$

$$(3)\ \frac{f(p,x_1,\ldots,x_n) \;=\; s \in q \qquad s \to s'}{f(s,t_1,\ldots,t_n) \to f(s',t_1,\ldots,t_n)}$$

$$(4)\ \frac{v \equiv v'}{\textbf{if } v{==}v' \textbf{ then } s \textbf{ else } s' \to s}$$

$$(5)\ \frac{v \not\equiv v'}{\textbf{if } v{==}v' \textbf{ then } s \textbf{ else } s' \to s'}$$

$$(6)\ \frac{t_1 \rightsquigarrow t_1'}{\textbf{if } t_1{==}t_2 \textbf{ then } s \textbf{ else } s' \to \textbf{if } t_1'{==}t_2 \textbf{ then } s \textbf{ else } s'}$$

$$(7)\ \frac{t \rightsquigarrow t'}{\textbf{if } v{==}t \textbf{ then } s \textbf{ else } s' \to \textbf{if } v{==}t' \textbf{ then } s \textbf{ else } s'}$$

$\boxed{t \rightsquigarrow t}$

$$(8)\ \frac{t \to t'}{t \rightsquigarrow t'}$$

$$(9)\ \frac{t \rightsquigarrow t'}{c(v_1,\ldots,v_{i-1},t,t_{i+1},\ldots,t_n) \rightsquigarrow c(v_1,\ldots,v_{i-1},t',t_{i+1},\ldots,t_n)}$$

$\boxed{t \Rightarrow_{cbn} t}$

$$(10)\ \frac{t \to t'}{t \Rightarrow_{cbn} t'}$$

$$(11)\ \frac{}{c(t_1,\ldots,t_n) \Rightarrow_{cbn} t_i}\ i \in \{1,\ldots,n\}$$

Figure 3.3: Small step, call-by-name semantics. $n, m \geq 0$.

Example 13
As an example of a reduction sequence, consider the term $f(add(\mathtt{s}(\mathtt{z}),\mathtt{z}))$ and the following program:

$$
\begin{array}{lcl}
\textbf{data}\ \mathrm{Bool}() & = & \textsc{true} \mid \textsc{false} \\
\textbf{data}\ \mathrm{Num}() & = & \mathtt{s}(\mathrm{Num}()) \mid \mathtt{z} \\
 & & \\
add(\mathtt{z},\mathtt{y}) & = & \mathtt{y} \\
add(\mathtt{s}(\mathtt{x}),\mathtt{y}) & = & \mathtt{s}(add(\mathtt{x},\mathtt{y})) \\
 & & \\
f(\mathtt{x}) & = & \textbf{if}\ add(\mathtt{x},\mathtt{s}(\mathtt{z}))\mathtt{==}\mathtt{s}(\mathtt{z})\ \textbf{then}\ \textsc{true}\ \textbf{else}\ \textsc{false}
\end{array}
$$

Evaluation of the term will proceed as follows:

$$
\begin{array}{ll}
 & f(add(\mathtt{s}(\mathtt{z}),\mathtt{z})) \\
\Rightarrow_{\mathrm{cbn}} & \textbf{if}\ add(add(\mathtt{s}(\mathtt{z}),\mathtt{z}),\mathtt{s}(\mathtt{z}))\mathtt{==}\mathtt{s}(\mathtt{z})\ \textbf{then}\ \textsc{true}\ \textbf{else}\ \textsc{false} \\
\Rightarrow_{\mathrm{cbn}} & \textbf{if}\ add(\mathtt{s}(add(\mathtt{z},\mathtt{z})),\mathtt{s}(\mathtt{z}))\mathtt{==}\mathtt{s}(\mathtt{z})\ \textbf{then}\ \textsc{true}\ \textbf{else}\ \textsc{false} \\
\Rightarrow_{\mathrm{cbn}} & \textbf{if}\ \mathtt{s}(add(add(\mathtt{z},\mathtt{z}),\mathtt{s}(\mathtt{z})))\mathtt{==}\mathtt{s}(\mathtt{z})\ \textbf{then}\ \textsc{true}\ \textbf{else}\ \textsc{false} \\
\Rightarrow_{\mathrm{cbn}} & \textbf{if}\ \mathtt{s}(add(\mathtt{z},\mathtt{s}(\mathtt{z})))\mathtt{==}\mathtt{s}(\mathtt{z})\ \textbf{then}\ \textsc{true}\ \textbf{else}\ \textsc{false} \\
\Rightarrow_{\mathrm{cbn}} & \textbf{if}\ \mathtt{s}(\mathtt{s}(\mathtt{z}))\mathtt{==}\mathtt{s}(\mathtt{z})\ \textbf{then}\ \textsc{true}\ \textbf{else}\ \textsc{false} \\
\Rightarrow_{\mathrm{cbn}} & \textsc{false}
\end{array}
$$

Chapter 4

Restriction Systems

During normal evaluation, all terms will be fully instantiated. This means that, for instance, testing whether two expressions are equal can be done in a simple way: first evaluate the first expression, then evaluate the second expression, and finally check whether the two values are identical.[1]

When we supercompile a program, however, we want to consider terms with free variables. When free variables are present in a term, we cannot in general *decide* whether two terms are equal because we have no knowledge about the value a particular variable will take; but we can find out whether it is *possible* that the terms are equal. Consider the term

$$\textbf{if } t{==}t' \textbf{ then } t_{\text{then}} \textbf{ else } t_{\text{else}}$$

When we supercompile this term, we want to eliminate one of the branches if it is possible to decide the outcome of condition, even in the case where we have imperfect information at hand.

We would therefore like to have a machinery into which we can insert equations like $t = t'$ and disequations[2] like $t \neq t'$, such that this machinery can tell us whether these equations and disequations can be satisfied, *i.e.* if there exists a solution to the system. An acceptable solution would be a substitution θ that ensures that every equation $t = t'$ is satisfied, *i.e.* that $t\theta$ is syntactically identical to $t'\theta$, and that every disequation $t \neq t'$ is satisfied, *i.e.* that $t\theta$ is syntactically different from $t'\theta$. In this chapter we will present such a machinery.

4.1 Restriction Systems

As we have seen in chapter 2, the strength of supercompilation lies in the way information is propagated while driving the program. We will use a *restriction system* inspired by [4, 14] to accumulate the information we gather about free variables. Each time new information is added to a restriction system, we can ask whether this newly obtained knowledge contradicts the information we have previously collected, *i.e.* it is possible to decide the *satisfiability* of such a system, a notion we will make precise later in this chapter.

A restriction system can thus be used to prune the process tree because it makes it possible to decide that a certain branch will never be executed.

Definition 14 (Restriction system)
A restriction system is a restricted kind of conjunctive normal form of certain formulae described in figure 4.1.

[1] Of course, if the evaluation of one of the expressions never terminates, we will not get an answer.

[2] The expression disequation is used to distinguish it from inequality like $x < y$.

$$
\begin{array}{llllll}
\text{(restriction systems)} & \mathscr{R} & \ni & \mathcal{R} & ::= & \mathcal{C}_1 \wedge \ldots \wedge \mathcal{C}_n \quad \text{(conjunction)} \\
 & & & \mathcal{C} & ::= & \mathcal{D}_1 \vee \ldots \vee \mathcal{D}_n \quad \text{(disjunction)} \\
 & & & & | & a = a' \quad \text{(equation)} \\
 & & & & | & \bot \quad \text{(false)} \\
 & & & & | & \top \quad \text{(true)} \\
 & & & \mathcal{D} & ::= & a \neq a' \quad \text{(disequation)} \\
 & & & & | & \bot \quad \text{(false)} \\
 & & & & | & \top \quad \text{(true)} \\
\text{(constructor terms)} & \mathscr{B} & \ni & a, b & ::= & x \quad \text{(variable)} \\
 & & & & | & c(a_1, \ldots, a_m) \quad \text{(constructor)}
\end{array}
$$

Figure 4.1: Restriction system grammar ($n > 0, m \geq 0$).

We work modulo commutativity of $\wedge$, $\vee$, $=$, and $\neq$; and modulo associativity of $\wedge$ and $\vee$. Therefore, we sometimes regard conjunctions and disjunctions as sets. Note also that the $\vee$-connective has precedence over the $\wedge$-connective, which is usually emphasised by parentheses. Substitutions are extended to restriction systems in the obvious way.

A restriction system only treats constructor terms. This is because the driving algorithm only need to propagate such information, as we shall see in the next chapter. We will now make precise what we mean by a solution to a restriction system.

Definition 15 (Solution)
The solution to a restriction system $\mathcal{R}$, denoted $\mathsf{Sol}(\mathcal{R})$, is the set of substitutions defined in the following way.

- $\mathsf{Sol}(\top) = \Theta$.
- $\mathsf{Sol}(\bigwedge_{i=1}^{n} \mathcal{C}_i) = \bigcap_{i=1}^{n} \mathsf{Sol}(\mathcal{C}_i)$.
- $\mathsf{Sol}(\bigvee_{i=1}^{n} \mathcal{D}_i) = \bigcup_{i=1}^{n} \mathsf{Sol}(\mathcal{D}_i)$.
- $\mathsf{Sol}(a = a') = \{\theta \mid a\theta \equiv a'\theta\}$.
- $\mathsf{Sol}(a \neq a') = \{\theta \mid a\theta \not\equiv a'\theta\}$.
- $\mathsf{Sol}(\bot) = \emptyset$.

If $\theta \in \mathsf{Sol}(\mathcal{R})$ we say that θ *validates* $\mathcal{R}$.

Our ultimate goal is to be able to decide whether there is a solution to a particular system. It turns out that this is easier if the system is simplified. We therefore introduce the notion of a *normal form* for a restriction system.

Definition 16 (Normal form)

1. A *solved variable* is a variable that occurs as one side of an equation and occurs exactly once in the system.

2. A *semi-solved variable* in a disjunction is a variable that occurs as one side of a disequation in a disjunction and occurs exactly once in this disjunction.

3. A restriction system in *normal form* is either $\bot$, $\top$, or of the form

$$\left(\bigwedge_{i=1}^{n} x_i = t_i\right) \wedge \bigwedge_{j=1}^{m} \left(\bigvee_{k=1}^{l} y_{j,k} \neq s_{j,k}\right)$$

where each x_i is a solved variable and each $y_{j,k}$ is a semi-solved variable.

Notice that a variable x may be semi-solved in one disjunction and not semi-solved in another disjunction. When we count the number of semi-solved variables, each disjunction is regarded as an isolated part of the system, *e.g.* the variable x counts as a semi-solved variable *and* as a not semi-solved variable.

The number of variables that are not solved or not semi-solved intuitively tells us how close we are to having normalised the system.

Example 17
Consider the following restriction system:

$$\begin{gathered} P(\text{CONS}(y, z_1), z_1) = P(x, z_2) \\ \wedge \\ P(y, z_3) \neq P(\text{CONS}(z_3, z_1), z_2) \end{gathered}$$

A normal form of this system could be

$$\begin{gathered} x = \text{CONS}(y, z_1) \\ \wedge \\ z_2 = z_1 \\ \wedge \\ y \neq \text{CONS}(z_3, z_3) \vee z_1 \neq z_3 \end{gathered}$$

in which x and z_2 are solved, and y and z_1 are semi-solved. Another normal form is

$$x = \text{CONS}(y, z_2) \wedge z_1 = z_2 \wedge (y \neq \text{CONS}(z_2, z_2) \vee z_3 \neq z_2)$$

where x and z_1 are solved, and y and z_3 are semi-solved. Hence there is in general no unique normal form for a restriction system.

Example 18
The restriction system $y \neq z \wedge (y \neq z \vee y \neq x)$ is in normal form. The sum of the number of variables that are not semi-solved is 1, since y is not semi-solved in $(y \neq z \vee y \neq x)$.

The restriction system $(y \neq z \vee y \neq x) \wedge (y \neq z \vee y \neq x)$ is in normal form. The sum of the number of variables that are not semi-solved is 2, since y is not semi-solved in both the disjunctions.

The rewrite rules defined in figure 4.2 transform a restriction system into a restriction system in normal form, which will be proved later.

Lemma 19
All rules in figure 4.2 are solution preserving, i.e. *if $\mathcal{R} \mapsto \mathcal{R}'$ then substitution θ validates $\mathcal{R}$ if and only if θ validates $\mathcal{R}'$.*

PROOF. Since most of the rewrite rules transform a local part of a system, we will implicitly use the fact that the context is left unchanged.

$$
\begin{array}{lrcll}
(T_1) & x = x & \mapsto & \top & \\
(T_2) & x \neq x & \mapsto & \bot & \\
(B_1) & \mathcal{C} \wedge \top & \mapsto & \mathcal{C} & \\
(B_2) & \mathcal{D} \vee \bot & \mapsto & \mathcal{D} & \\
(B_3) & \mathcal{C} \wedge \bot & \mapsto & \bot & \\
(B_4) & \mathcal{D} \vee \top & \mapsto & \top & \\
(C_1) & c(b_1, \ldots, b_n) = c'(a_1, \ldots, a_m) & \mapsto & \bot & (c \neq c') \\
(C_2) & c(b_1, \ldots, b_n) \neq c'(a_1, \ldots, a_m) & \mapsto & \top & (c \neq c') \\
(D_1) & c(b_1, \ldots, b_n) = c(a_1, \ldots, a_n) & \mapsto & b_1 = a_1 \wedge \ldots \wedge b_n = a_n & (n \geq 1) \\
(D_2) & c(b_1, \ldots, b_n) \neq c(a_1, \ldots, a_n) & \mapsto & b_1 \neq a_1 \vee \ldots \vee b_n \neq a_n & (n \geq 1) \\
(D_3) & c = c & \mapsto & \top & \\
(D_4) & c \neq c & \mapsto & \bot & \\
(O_1) & x = a & \mapsto & \bot & (x \in \mathsf{Var}(a) \,\&\, a \not\equiv x) \\
(O_2) & x \neq a & \mapsto & \top & (x \in \mathsf{Var}(a) \,\&\, a \not\equiv x) \\
(R_1) & x = a \wedge \bigwedge_{i=1}^{n} \mathcal{C}_i & \mapsto & x = a \wedge \bigwedge_{i=1}^{n} \mathcal{C}_i\{x := a\} & \\
(R_2) & x \neq a \vee \bigvee_{i=1}^{n} \mathcal{D}_i & \mapsto & x \neq a \vee \bigvee_{i=1}^{n} \mathcal{D}_i\{x := a\} &
\end{array}
$$

Additional control on rules:

(R_1) $x \in \bigcup \mathsf{Var}(\mathcal{C}_i)$ and $x \notin \mathsf{Var}(a)$. If a is a variable y then $y \in \bigcup \mathsf{Var}(\mathcal{C}_i)$. This rule rewrites the *whole* system.

(R_2) $x \in \bigcup \mathsf{Var}(\mathcal{D}_i)$ and $x \notin \mathsf{Var}(a)$. If a is a variable y then $y \in \bigcup \mathsf{Var}(\mathcal{D}_i)$. This rule rewrites the *whole* disjunction.

Figure 4.2: Rewrite rules for constraints.

(T_1) $x = x \mapsto \top$: $\mathsf{Sol}(x = x) = \Theta = \mathsf{Sol}(\top)$.

(T_2) $x \neq x \mapsto \bot$: $\mathsf{Sol}(x \neq x) = \emptyset = \mathsf{Sol}(\bot)$.

(B_1) $\mathcal{C} \wedge \top \mapsto \mathcal{C}$: $\mathsf{Sol}(\mathcal{C} \wedge \top) = \mathsf{Sol}(\mathcal{C}) \cap \mathsf{Sol}(\top) = \mathsf{Sol}(\mathcal{C})$.

(B_2) $\mathcal{D} \vee \bot \mapsto \mathcal{D}$: $\mathsf{Sol}(\mathcal{D} \vee \bot) = \mathsf{Sol}(\mathcal{D}) \cup \mathsf{Sol}(\bot) = \mathsf{Sol}(\mathcal{D})$.

(B_3) $\mathcal{C} \wedge \bot \mapsto \bot$: $\mathsf{Sol}(\mathcal{C} \wedge \bot) = \mathsf{Sol}(\mathcal{C}) \cap \mathsf{Sol}(\bot) = \mathsf{Sol}(\bot)$.

(B_4) $\mathcal{D} \vee \top \mapsto \top$: $\mathsf{Sol}(\mathcal{D} \vee \top) = \mathsf{Sol}(\mathcal{D}) \cup \mathsf{Sol}(\top) = \mathsf{Sol}(\top)$.

(C_1) $\mathcal{R} \equiv c(a_1, \ldots, a_n) = c'(a_1, \ldots, a_m) \mapsto \mathcal{R}' \equiv \bot$: Since $c \neq c'$, no substitution will validate $\mathcal{R}$ or $\mathcal{R}'$.

(C_2) $\mathcal{R} \equiv c(a_1, \ldots, a_n) \neq c'(a_1, \ldots, a_m) \mapsto \mathcal{R}' \equiv \top$: Since $c \neq c'$, any substitution will validate both $\mathcal{R}$ and $\mathcal{R}'$.

(D_1) $\mathcal{R} \equiv (c(b_1, \ldots, b_n) = c(a_1, \ldots, a_n)) \mapsto b_1 = a_1 \wedge \cdots \wedge b_n = a_n \equiv \mathcal{R}'$: Any substitution θ that validates $\mathcal{R}$ would have to make each subcomponent $b_i\theta \equiv a_i\theta$. Thus it would also validate $\mathcal{R}'$. Conversely, the above applies backwards.

(D_2) $\mathcal{R} \equiv (c(b_1, \ldots, b_n) \neq c(a_1, \ldots, a_n)) \mapsto b_1 \neq a_1 \vee \cdots \vee b_n \neq a_n \equiv \mathcal{R}'$: Any substitution θ that validates $\mathcal{R}$ would have to validate at least one subcomponent such that $b_i\theta \not\equiv a_i\theta$. Thus it would also validate $\mathcal{R}'$. Conversely, the above applies backwards.

(D_3) $c = c \mapsto \top$: $\mathsf{Sol}(c = c) = \Theta = \mathsf{Sol}(\top)$.

(D_4) $c \neq c \mapsto \bot$: $\mathsf{Sol}(c \neq c) = \emptyset = \mathsf{Sol}(\bot)$.

(O_1) $x = a \mapsto \bot$: Since $x \in \mathsf{Var}(a)$, no substitution will validate $x = a$ or $\bot$.

(O_2) $x \neq a \mapsto \top$: Since $x \in \mathsf{Var}(a)$, any substitution will validate both $x \neq a$ and $\top$.

(R_1) $\mathcal{R} \equiv (x = a \wedge \bigwedge_{i=1}^{n} \mathcal{C}_i) \mapsto (x = a \wedge \bigwedge_{i=1}^{n} \mathcal{C}_i\{x{:=}a\}) \equiv \mathcal{R}'$:

($\Rightarrow$) Suppose substitution θ validates $\mathcal{R}$. Then $x\theta \equiv a\theta$. For each $\mathcal{C}_i$ of the form $a_i = a_i'$, it will be the case that

$$\begin{array}{rcl} \mathcal{C}_i\theta & \equiv & a_i\theta = a_i'\theta \\ & \equiv & a_i\{x{:=}a\}\theta = a_i'\{x{:=}a\}\theta \quad \text{(since } x\theta \equiv a\theta \text{ and } \theta \text{ is idempotent)} \\ & \equiv & \mathcal{C}_i\{x{:=}a\}\theta \end{array}$$

For each $\mathcal{C}_i$ of the form $\bigvee_{j=1}^{m} \mathcal{D}_j$, it will be the case that

$$\begin{array}{rcl} \mathcal{D}_j\theta & \equiv & a_j\theta \neq a_j'\theta \\ & \equiv & a_j\{x{:=}a\}\theta \neq a_j'\{x{:=}a\}\theta \quad \text{(since } x\theta \equiv a\theta \text{ and } \theta \text{ is idempotent)} \\ & \equiv & \mathcal{D}_j\{x{:=}a\}\theta \end{array}$$

Thus $\mathcal{R}\theta \equiv \mathcal{R}'\theta$, so θ also validates $\mathcal{R}'$.

($\Leftarrow$) The above reasoning applies backwards.

(R_2) $\mathcal{R} \equiv (x \neq a \vee \bigvee_{i=1}^{n} \mathcal{D}_i) \mapsto (x \neq a \vee \bigvee_{i=1}^{n} \mathcal{D}_i\{x{:=}a\}) \equiv \mathcal{R}'$:

($\Rightarrow$) Any substitution θ that validates $\mathcal{R}$ would either make $x\theta \not\equiv a\theta$ or $x\theta \equiv a\theta$. In the former case we trivially have that θ validates $\mathcal{R}'$. In the latter case, the proof is similar to that of rule R_1.

($\Leftarrow$) As above with $\mathcal{R}$ and $\mathcal{R}'$ swapped.

□

Lemma 20
Non-deterministic, exhaustive application of the rewrite rules in figure 4.2 to any restriction system terminates.

PROOF. It is sufficient to find a function that, when applied on any restriction system, decreases with every application of any rewrite rule.

Let a function Φ have the set of restriction systems as domain and as range ordered tuples of type $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$. Let Φ be defined by three functions ϕ_1, ϕ_2, ϕ_3 such that $\Phi(\mathcal{R}) = (\phi_1(\mathcal{R}), \phi_2(\mathcal{R}), \phi_3(\mathcal{R}))$:

$\phi_1(\mathcal{R})$ is the number of variables in $\mathcal{R}$ that are not solved.

$\phi_2(\mathcal{R})$ is the sum of the number of variables in $\mathcal{R}$ that are not semi-solved in each disjunction (see example 18).

$\phi_3(\mathcal{R})$ is size($\mathcal{R}$), which is the number of variables, constructors, $\bot$s, and $\top$s in $\mathcal{R}$.

Notice that no rule introduces new variables and that no variable can be both solved and semi-solved.

Each rule decreases the ordered tuple:

Rules T_1, T_2, B_1–$B_4, C_1, C_2, D_3, D_4, O_1, O_2$: The rules possibly eliminate variables, in which case neither ϕ_1 nor ϕ_2 increases. In any case, the size of the system decreases, and thus ϕ_3 decreases.

Rule D_1: No disjunction is affected, so ϕ_2 does not increase. No solved variable can be changed into a not solved one, since a solved variable only appears once in a system, so ϕ_1 do not increase. In any case, the size of the system decreases with the disposal of c, so ϕ_3 decreases.

Rule D_2: Similar to D_1.

Rule R_1:

1. If a is not a variable y, no variable $z \in \mathsf{Var}(a)$ can be a solved variable, and thus the duplication of a does not increase ϕ_1.

2. If a is a variable y, y is not a solved variable since $y \in \bigcup \mathsf{Var}(\mathcal{C}_i)$, and thus the duplication of a does not increase ϕ_1.

In any case, x is not a solved variable since $x \in \bigcup \mathsf{Var}(\mathcal{C}_i)$. However, x becomes a semi-solved variable and hence ϕ_2 decreases.

Hence the ordered tuple decreases with every application of the rewrite rules. □

Definition 21 ($\mapsto\!\!\!\!\!\rightarrow$)
Non-deterministic, exhaustive application of the rewrite rules in figure 4.2 is denoted $\mapsto\!\!\!\!\!\rightarrow$.

We thus write $\mathcal{R} \mapsto\!\!\!\!\!\rightarrow \mathcal{R}'$ if the exhaustive application of the rewrite rules to $\mathcal{R}$ results in $\mathcal{R}'$.

Lemma 22
The non-deterministic, exhaustive application of the rewrite rules in figure 4.2 to any restriction system results in a restriction system in normal form.

PROOF. By lemma 20, any exhaustive application of the rewrite rules terminates and results in a restriction system $\mathcal{R}$. It thus sufficient to prove that, when no rewrite rule applies to $\mathcal{R}$, $\mathcal{R}$ is in normal form. We prove the contrapositive: when $\mathcal{R}$ is not in normal form, some rewrite rule applies.

Assume that $\mathcal{R}$ is not in normal form, which means that $\mathcal{R} \not\equiv \bot$ and $\mathcal{R} \not\equiv \top$. Then either

- $\mathcal{R}$ contains $\bot$ or $\top$. Then rules B_1–B_4 apply.
- $\mathcal{R}$ contains an equation or a disequation where both sides are non-variables. Then rules C_1, C_2 or D_1–D_4 apply.
- $\mathcal{R}$ contains an equation $x = a$ where neither x nor a is a solved variable.
  1. If $x \in \mathsf{Var}(a)$ then either rule T_1 or O_1 apply.
  2. If $x \notin \mathsf{Var}(a)$, then x must occur elsewhere in the system. Then either
     (a) a is not a variable, in which case rule R_1 applies, or
     (b) a is a variable y, in which case y would not be a solved variable so rule R_1 applies.
- $\mathcal{R}$ contains a disequation $x \neq a$ neither x nor a is a semi-solved variable.
  1. If $x \in \mathsf{Var}(a)$ then either rule T_2 or O_2 apply.
  2. If $x \notin \mathsf{Var}(a)$, then x must occur elsewhere in the disjunction. Then either
     (a) a is not a variable, in which case rule R_2 applies, or
     (b) a is a variable y, in which case y would not be a semi-solved variable so rule R_2 applies.

□

4.2 Satisfiability

With the rewrite rules that transform a restriction system to normal form, we are now much closer to being able to decide whether a restriction system is *satisfiable*. For instance, in the case of the restriction system $\mathrm{P}(\mathrm{TRUE}, x) = \mathrm{P}(\mathrm{FALSE}, y)$, exhaustive application of the rewrite rules will always result in $\bot$, so we can say with no doubt that the solution is the empty set, and thus the system cannot be satisfiable. We will now, however, illustrate that the existence of a non-empty *solution* is not enough to make the a restriction system *satisfiable*.

Ideally, we would like to decide whether a restriction system is satisfiable by a mere syntactic inspection of a normal form of the restriction system, *e.g.* a system is satisfiable if, and only if, the system reduces to anything but $\bot$, but this is not possible: consider the restriction system $x \neq y \wedge y \neq z \wedge z \neq x$, which is in normal form. Since this system is different from $\bot$, we would suspect that it is satisfiable. However, if we know that variables x, y and z all[3] have boolean type, it turns out that it is impossible to assign *values* (*i.e.* TRUE or FALSE) to the three variables in a way that satisfies the system, even though the substitution with the empty support will *validate*

[3] Since programs must be type correct, both sides of equations and disequations will have the same type.

the system. To realise this, our machinery has to systematically try out all possible combinations of value assignments for variables. If the three variables are of type "list of booleans", however, it will always be possible to assign values to the three variables such that the system is satisfied, since there are infinitely many lists of booleans.

Remark 23 The problems illustrated above can be described informally as a "shortage" of values: in some situations there are simply not enough values to satisfy the disequalities in a particular restriction system. Equations, on the other hand, represent somewhat more absolute information: if a normalised restriction system contains an equation $\mathfrak{x} = \mathfrak{a}$, values can easily be assigned; we simply assign arbitrary values to the variables in $\mathfrak{a}$, which will then gives us a value for $\mathfrak{x}$ since $\mathfrak{x}$ does not occur elsewhere in the system. The only case where this is not possible is when the type of $\mathfrak{a}$ (and thus of $\mathfrak{x}$) has no values at all. This problem can be overcome by requiring that every type definition in a program must have at least one value.

So far, we have dealt with terms containing free variables without requiring anything about values such free variables can assume. This seems quite reasonable when we consider our application: supercompilation should be able to deal with imperfect information, as illustrated in the example in chapter 2. Consider again the term $a(\mathfrak{xs}, \mathfrak{ys})$, where *a* is the append function. It is obvious that the variables $\mathfrak{xs}$ and $\mathfrak{ys}$ are of some list type, but we cannot tell what type the elements are, *i.e.* what values the elements can assume.

We will now assume that variables are annotated with *type information* that will tell us whether a particular variable ranges over a finite set of values. If we recall the definition of a finite type from chapter 3, we can now divide the variables into two partitions: those that have finite type and those that do not.

Definition 24 (Variable of finite type)
A variable of *finite type* can only assume a finite set of values.

We are now able to introduce an *explosion rule* that takes care of the situation where we have special knowledge about which values a variable can assume.

Definition 25 (Explosion rule)
The *explosion rule* is defined thus:

$$\mathcal{R} \hookrightarrow \mathcal{R}\{\mathfrak{x} := \nu\} \wedge \mathfrak{x} = \nu$$

where ν is a value of the same type as $\mathfrak{x}$. The explosion rule can be applied to $\mathcal{R}$ if and only if

1. $\mathcal{R}$ is in normal form, and
2. $\mathcal{R}$ contains a disequation $\mathfrak{x} \neq \mathfrak{a}$, and
3. The type of $\mathfrak{x}$ is finite.

Example 26
Let the restriction system $\mathcal{R}$ be

$$\mathfrak{x} \neq \mathfrak{y} \wedge \mathfrak{y} \neq z \wedge z \neq \mathfrak{x}$$

where $\mathfrak{x}, \mathfrak{y}$ and z all have boolean type as above. The explosion rule applies to $\mathcal{R}$ and could result in

$$\text{TRUE} \neq \mathfrak{y} \wedge \mathfrak{y} \neq z \wedge z \neq \text{TRUE} \wedge \mathfrak{x} = \text{TRUE}$$

If we continue to use the explosion rule in combination with the other rewrite rules, we could see the following development.

$$\begin{array}{c}
\text{TRUE} \neq y \wedge y \neq z \wedge z \neq \text{TRUE} \wedge x = \text{TRUE} \\
\hookrightarrow \\
\text{TRUE} \neq \text{FALSE} \wedge \text{FALSE} \neq z \wedge z \neq \text{TRUE} \wedge x = \text{TRUE} \wedge y = \text{FALSE} \\
\mapsto \\
\text{FALSE} \neq z \wedge z \neq \text{TRUE} \wedge x = \text{TRUE} \wedge y = \text{FALSE} \\
\hookrightarrow \\
\text{FALSE} \neq \text{FALSE} \wedge \text{FALSE} \neq \text{TRUE} \wedge x = \text{TRUE} \wedge y = \text{FALSE} \wedge z = \text{FALSE} \\
\mapsto \\
\bot
\end{array}$$

It turns out that any such development will result in $\bot$, which is exactly what we were looking for, since we can easily distinguish $\bot$ from any other restriction system in normal form.

The satisfiability of a restriction system can now be formally defined by requiring that it must be possible to assign a value to every variable with finite type. For this end we will define a special kind of substitution.

Definition 27 (Value substitution)
A *value substitution* $\underline{\theta}$ is a partial map from variables to values where, for each $x \in \mathsf{support}(\underline{\theta})$, x and $x\underline{\theta}$ must have the same type.

Definition 28 (Satisfiable)
Given a restriction system $\mathcal{R}$,

1. $\mathcal{R}$ is *satisfiable* if and only if there exists a value substitution $\underline{\rho}$ with support $\{x \mid x \in \mathsf{Var}(\mathcal{R})$ and the type of x is finite$\}$ and a substitution θ such that $\theta \oplus \underline{\rho}$ validates $\mathcal{R}$. We will call $\theta \oplus \underline{\rho}$ a *satisfier of* $\mathcal{R}$.

2. $\mathcal{R}$ is *trivially satisfied by* $\underline{\rho}$ if and only if $\mathsf{support}(\underline{\rho}) = \mathsf{Var}(\mathcal{R})$ and $\theta \oplus \underline{\rho}$ validates $\mathcal{R}$, for any substitution θ.

Example 29
Consider the restriction system $x = \text{P}(y, z_1) \wedge z_1 \neq z_2 \wedge z_3 \neq z_2$ where z_1, z_2, z_3 are of boolean type and y is of unknown type. The system is satisfiable because the substitution $\{x := \text{P}(y, \text{TRUE})\} \oplus \{z_1 := \text{TRUE}, z_2 := \text{FALSE}, z_3 := \text{TRUE}\}$ will validate the system.

Lemma 30
If the explosion rule applies to a restriction system $\mathcal{R}$, it preserves satisfiability : A substitution $\theta \oplus \underline{\rho}$ is a satisfier for $\mathcal{R}$ if and only if $\theta \oplus \underline{\rho}$ is a satisfier for at least one $\mathcal{R}'$, where $\mathcal{R} \hookrightarrow \mathcal{R}'$.

PROOF. Assume that $\mathcal{R} \hookrightarrow \mathcal{R}\{x := v\} \wedge x = v$ where x has finite type.

If $\theta \oplus \underline{\rho}$ is a satisfier for $\mathcal{R}$, it means that $x \in \mathsf{support}(\underline{\rho})$ since x has finite type. Thus $\theta \oplus \underline{\rho}$ is also a satisfier for one $\mathcal{R}\{x := v_i\} \wedge x = v_i$, namely one where $x(\theta \oplus \underline{\rho}) \equiv v_i$.

Conversely, if $\theta \oplus \underline{\rho}$ is a satisfier for $\mathcal{R}\{x := v\} \wedge x = v$, it is also a satisfier for $\mathcal{R}$ since $x(\theta \oplus \underline{\rho}) \equiv v$. □

We would now like to prove that the rewrite system in figure 4.2 preserves satisfiability, but this is not strictly true. The reason for this is that a satisfier for some restriction system $\mathcal{R}$ is chained to $\mathcal{R}$ in the sense that it is required to map all variables of finite type *in* $\mathcal{R}$ to a value.

Since some of the rewrite rules in figure 4.2 possibly eliminate variables from the restriction system, the rewrite system is not *sound* with regard to satisfiability, *i.e.* the set of satisfiers that validate the system before the rewrite might be extended with unexpected satisfiers after the rewrite.

Example 31
Consider the restriction system $\mathcal{R} \equiv x = x$, where x is of finite type. By inference rule T1, $\mathcal{R}$ rewrites to $\top$. A substitution θ that is a satisfier for $\mathcal{R}$ is also a satisfier for $\top$, since any substitution is a satisfier for $\top$. However, the substitution $\theta_\emptyset$ with empty support is a satisfier for $\top$, but $\theta_\emptyset$ is not a satisfier for $\mathcal{R}$, since a satisfier for $\mathcal{R}$ must map x to a value.

As suggested by this example, the rewrite system in figure 4.2 is *complete* with regard to satisfiability, *i.e.* a satisfier that validates the system before the rewrite will also validate the system that is obtained by the rewrite.

Lemma 32 ($\mapsto$ is complete)
The rewrite system in figure 4.2 is complete with regard to satisfiability.

PROOF. Assume $\mathcal{R} \mapsto \mathcal{R}'$. If θ is a satisfier for $\mathcal{R}$ it will validate $\mathcal{R}$. By lemma 19, θ will also validate $\mathcal{R}'$. □

We thus know that, given a restriction system $\mathcal{R}$, any sequence of rewrite steps (including explosion) will be complete with regard to satisfiability. This means that if $\mathcal{R}$ is satisfiable, then at least one of the restriction systems that result from the rewrite steps is also satisfiable. Conversely, if none of the restriction systems that result from the rewrite steps are satisfiable, then $\mathcal{R}$ cannot be satisfiable. This last property can then be used by our machinery: if $\mathcal{R}$ by any rewrite sequence results in an unsatisfiable system, the question "is $\mathcal{R}$ satisfiable?" can definitely be answered by "no!". The only remaining problem is now how to known when a restriction system to which no rewrite rule applies is unsatisfiable.

Proposition 33
Given a restriction system $\mathcal{R}$ in normal form to which the explosion rule does not apply, $\mathcal{R}$ is satisfiable if and only if $\mathcal{R} \not\equiv \bot$.

PROOF. ($\Rightarrow$) Assume $\mathcal{R}$ is satisfiable. By definition $\mathcal{R}$ is not $\bot$.

($\Leftarrow$) Assume $\mathcal{R}$ is not $\bot$. We will now enumerate the different forms of $\mathcal{R}$. Let the trivial substitution with empty support be denoted $\theta_\emptyset$.

$\mathcal{R} \equiv \top$**:** Since $\mathcal{R}$ contains no variables, the trivial substitution $\theta_\emptyset$ is a satisfier for $\mathcal{R}$.

$\mathcal{R} \equiv \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m} x_{i,j} \neq a_{i,j}$**:** Given a value substitution $\underline{\rho}$ with support $\{y \mid y \in \mathsf{Var}(\mathcal{R}) \,\&\, \text{type of } y \text{ is finite}\}$. For all i: given some j, consider the disequation $x_{i,j} \neq a_{i,j}$. Either

1. $a_{i,j}$ is a variable z. Since $\mathcal{R}$ is in normal form, z is different from $x_{i,j}$. By assumption the explosion rule does not apply, and thus neither $x_{i,j}$ nor z is a variable of finite type, and then $\mathsf{support}(\underline{\rho}) \cap \{x_{i,j}, z\} = \emptyset$. Hence $\theta_\emptyset \oplus \underline{\rho}$ will validate $x_{i,j} \neq a_{i,j}$.
2. $a_{i,j}$ is not a variable. By assumption the explosion rule does not apply, and thus $x_{i,j}$ is not a variable of finite type, and then $x_{i,j} \notin \mathsf{support}(\underline{\rho})$. Hence $\theta_\emptyset \oplus \underline{\rho}$ will validate $x_{i,j} \neq a_{i,j}$.

Hence $\theta_\emptyset \oplus \underline{\rho}$ will validate $\mathcal{R}$.

$\mathcal{R} \equiv \bigwedge_{i=1}^{n} x_i = a_i$: Given a value substitution $\underline{\rho}$ with support $\{y \mid y \in \mathsf{Var}(\mathcal{R}) \,\&\, \text{type of } y \text{ is finite}\} \setminus \{x_1, \ldots, x_n\}$. For all i, either

1. x_i has finite type. Then every variable in a_i has finite type. Since $\mathcal{R}$ is in normal form, $x_i \notin \mathsf{Var}(a_i)$. Thus $\mathsf{Var}(a_i) \subseteq \mathsf{support}(\underline{\rho})$. Let now the value substitution $\underline{\rho}_i$ defined by

$$x\underline{\rho}_i = \begin{cases} x\underline{\rho} & \text{if } x \in \mathsf{support}(\underline{\rho}) \\ a_i\underline{\rho} & \text{if } x = x_i \\ \text{undefined} & \text{otherwise} \end{cases}$$

   Then $\theta_\emptyset \oplus \underline{\rho}_i$ will validate $x_i = a_i$.

2. x_i does not have finite type. Since $\mathcal{R}$ is in normal form, $x_i \notin \mathsf{Var}(a_i)$. Since $x_i \notin \mathsf{support}(\underline{\rho})$, $\{x_i := a_i\underline{\rho}\} \oplus \underline{\rho}$ will validate $x_i = a_i$.

We can now construct a substitution and a value substitution that validates $\mathcal{R}$: since every x_i occurs exactly once in the system,

1. for each i where x_i has finite type, $\underline{\rho}_i$ only differs from $\underline{\rho}$ by including x_i in its support. We can thus build a value substitution $\underline{\rho}'$ from all such $\underline{\rho}_i$ such that the support of $\underline{\rho}'$ includes all variables with finite type in $\mathcal{R}$.
2. from all i where x_i does not have finite type, we can build a substitution θ such that the support of θ includes all x_i that does not have finite type in $\mathcal{R}$.

Hence $\theta \oplus \underline{\rho}'$ will validate $\mathcal{R}$.

$\mathcal{R} \equiv \bigwedge_{i=1}^{n} x_i = a_i \wedge \bigwedge_{k=1}^{l} \bigvee_{j=1}^{m} x_{k,j} \neq a_{k,j}$: From the case above, we can construct a substitution θ and a value substitution $\underline{\rho}'$ such that the part of $\mathcal{R}$ that only contains equations can be validated. We will now extend the support of the value substitution such that the extended value substitution together θ will validate $\mathcal{R}$.

Let θ and $\underline{\rho}'$ be constructed as in the case above and let $X = \{y \mid y \in \mathsf{Var}(\mathcal{R}) \,\&\, \text{type of } y \text{ is finite}\} \setminus \mathsf{support}(\underline{\rho})$. Now define the value substitution $\underline{\rho}''$ as follows:

$$x\underline{\rho}'' = \begin{cases} x\underline{\rho}' & \text{if } x \in \mathsf{support}(\underline{\rho}') \\ v & \text{if } x \in X, \text{where } v \text{ has the same type as } x \\ \text{undefined} & \text{otherwise} \end{cases}$$

Since $\underline{\rho}''$ only differs from $\underline{\rho}'$ by including variables that are not present in the part of $\mathcal{R}$ that only contains equations, $\theta \oplus \underline{\rho}''$ will validate the part of $\mathcal{R}$ that only contains equations. Furthermore, since 1) the support of $\underline{\rho}''$ includes all variables of finite type in $\mathcal{R}$, and 2) the support of θ does not include any variables in the part of $\mathcal{R}$ that only contains disequations, $\theta \oplus \underline{\rho}''$ will also validate the part of $\mathcal{R}$ that only contains disequations. Hence $\theta \oplus \underline{\rho}''$ will validate $\mathcal{R}$.

□

With the explosion rule we can define an algorithm $\mathsf{satisfiable} : \mathscr{R} \to \{\text{TRUE}, \text{FALSE}\}$ that calculates the satisfiability of a restriction system.

Algorithm 34

$$
\begin{array}{lcl}
\mathsf{satisfiable}(\mathcal{R}) & = & \textbf{let } \mathcal{R} \rightarrowtail\!\!\!\!\!\twoheadrightarrow \mathcal{R}' \\
& & \textbf{in if } \mathcal{R}' \text{ contains a disequation } x \neq a \text{ where the type of } x \text{ is finite} \\
& & \textbf{then let } \{\mathcal{R}_1, \ldots, \mathcal{R}_n\} = \{\mathcal{R}'\{x{:=}v\} \wedge x = v \mid \mathcal{R}' \hookrightarrow \mathcal{R}'\{x{:=}v\} \wedge x = v\} \\
& & \qquad \textbf{in } \bigvee_{i=1}^{n} \mathsf{satisfiable}(\mathcal{R}_i) \\
& & \textbf{else if } \mathcal{R}' \equiv \bot \textbf{ then } \textsc{false} \textbf{ else } \textsc{true}
\end{array}
$$

Proposition 35
The application of algorithm satisfiable to any restriction system terminates.

PROOF. The initial normalisation of the restriction system terminates by lemma 20. The test on whether the explosion rule applies clearly terminates, since the restriction system is finite and thus can be traversed.

1. The else-branch of the test clearly terminates.
2. In the then-branch, the result of the explosion rule is a finite set, since the chosen variable x is of finite type. Each restriction system in this set will have one more solved variable than $\mathcal{R}'$, and $\mathcal{R}'$ will have at least as many solved variables as $\mathcal{R}$.

Hence each invocation of the satisfiable function will either terminate immediately or increase the number of solved variables for each recursive invocation, which cannot go on indefinitely since the number of variables in a restriction system is finite. □

Lemma 36
The algorithm satisfiable applied to any restriction system $\mathcal{R}$ evaluates to TRUE *if $\mathcal{R}$ is satisfiable.*

PROOF. Assume that $\mathcal{R}$ is satisfiable, which means that there exists a satisfier θ that validates $\mathcal{R}$.

The algorithm will rewrite and explode $\mathcal{R}$ until no rewrite or explosion rule applies. By lemma 30 and lemma 32, we know that any sequence of rewrites will be complete with regard to satisfiability, and thus θ will be a satisfier for at least one restriction system that results from a series of rewrite and explosion steps. By lemma 35 we know that the algorithm terminates and thus it must reach a restriction system $\mathcal{R}'$ in normal form to which the explosion rule does not apply and for which θ is a satisfier. By proposition 33, we know that if the explosion rule does not apply to such $\mathcal{R}'$ and $\mathcal{R}'$ is satisfiable, $\mathcal{R}'$ is different from $\bot$. Thus the algorithm will return TRUE. □

We have thus developed a machinery that can decide whether a set of constraints can be satisfied. If we are concerned with the complexity of resolving satisfiability (which is NP-complete), we can refrain from using the explosion rule (and thus we need not require that variables are type annotated). If the explosion rule is left out, we will still have a safe approximation of the satisfiability. It remains to be seen whether such an approximation will work in practice.

4.3 Extraction of Substitutions

Previous definitions of supercompilers in a setting similar to our [24, 27, 25, 26], utilised positive information by means of substitutions calculated by unification. It has been noted in [4] that

normalisation of restriction systems encompasses unification, which suggests that it must be possible to extract substitutions after normalisation of restriction systems. As we will see in the next chapter, extracting such substitutions will be highly valuable. We will therefore define an extraction operation on restriction systems.

Definition 37 (Extract)
The function $\mathsf{extract} : \mathscr{R} \to \Theta \times \mathscr{R}$ normalises a restriction system and divides it into two parts: a substitution θ induced by all equations, and a new restriction system that consists of all disjunctions.

Lemma 38 (Substitution from restriction system)
A substitution can always be created from the equational part of a restriction system in normal form different from $\bot$.

PROOF. Since the restriction system is in normal form, it is either

1. $\top$, in which case we define the extracted substitution to have an empty support, or
2. it contains equations and/or disequations: each equation contains a solved variable x. This implies that x is not present elsewhere in the system. Thus we can construct a substitution such that the set of solved variables constitutes the support of the substitution and the other sides of the equations constitute the range.

□

After substitution-extraction, the remaining part of the restriction system contains negative information only, *i.e.* a conjunction of disjunctions of disequations.

Definition 39 (Negative system)
A *negative system* is a restriction system in normal form that only contains negative information, *i.e.* a conjunction of disjunctions of disequations.

Example 40
Consider the again the restriction system from example 17:

$$\mathrm{P}(\mathrm{CONS}(y, z_1), z_1) = \mathrm{P}(x, z_2) \wedge \mathrm{P}(y, z_3) \neq \mathrm{P}(\mathrm{CONS}(z_3, z_1), z_2)$$

A normal form of this system could be

$$x = \mathrm{CONS}(y, z_1) \wedge z_2 = z_1 \wedge (y \neq \mathrm{CONS}(z_3, z_3) \vee z_1 \neq z_3)$$

From this normal form we can extract the substitution $\{x := \mathrm{CONS}(y, z_1), z_2 := z_1\}$ and the negative system $y \neq \mathrm{CONS}(z_3, z_3) \vee z_1 \neq z_3$.

Lemma 41 (MGU)
A constraint system $a = a'$ normalises to the most general unifier of a and a' , denoted mgu(a, a').

PROOF. For a system that only contains equations, rules T1, B1, B3, C1, D1, D3, O1, and R1 are the only rules that apply. These rules are equivalent to the unification algorithm defined in [17] when we interpret $\bot$ as failure and otherwise extract the substitution as defined in the proof of lemma 38. □

4.4 Comparing Negative Systems

In the next chapter we will need to compare negative systems with regard to their solutions: given two constraint systems $\mathcal{R}, \mathcal{R}'$, we want to decide whether $\mathsf{Sol}(\mathcal{R}') \subseteq \mathsf{Sol}(\mathcal{R})$. The following lemmas will establish a set of conditions that, when satisfied, will make this relation hold.

Lemma 42
Given $\mathcal{R} \equiv \bigwedge_{i=1}^{n} \mathcal{C}_i$ *and* $\mathcal{R}' \equiv \bigwedge_{k=1}^{m} \mathcal{C}'_k$.

$$\begin{array}{c} \forall i \in \{1,\ldots,n\} : \exists j_i \in \{1,\ldots,m\} : \mathit{Sol}(\mathcal{C}'_{j_i}) \subseteq \mathit{Sol}(\mathcal{C}_i) \\ \Downarrow \\ \mathit{Sol}(\mathcal{R}') \subseteq \mathit{Sol}(\mathcal{R}) \end{array}$$

PROOF.

$$\mathsf{Sol}(\mathcal{R}') = \bigcap_{k=1}^{m} \mathsf{Sol}(\mathcal{C}'_k) \subseteq \bigcap_{i=1}^{n} \mathsf{Sol}(\mathcal{C}'_{j_i}) \subseteq \bigcap_{i=1}^{n} \mathsf{Sol}(\mathcal{C}_i) = \mathsf{Sol}(\mathcal{R})$$

□

Lemma 43
Given $\mathcal{C} \equiv \bigvee_{i=1}^{n} \mathcal{D}_i$ *and* $\mathcal{C}' \equiv \bigvee_{k=1}^{m} \mathcal{D}'_k$.

$$\begin{array}{c} \forall k \in \{1,\ldots,m\} : \exists j_k \in \{1,\ldots,n\} : \mathit{Sol}(\mathcal{D}'_k) \subseteq \mathit{Sol}(\mathcal{D}_{j_k}) \\ \Downarrow \\ \mathit{Sol}(\mathcal{C}') \subseteq \mathit{Sol}(\mathcal{C}) \end{array}$$

PROOF.

$$\mathsf{Sol}(\mathcal{C}') = \bigcup_{k=1}^{m} \mathsf{Sol}(\mathcal{D}'_k) \subseteq \bigcup_{k=1}^{m} \mathsf{Sol}(\mathcal{D}_{j_i}) \subseteq \bigcup_{i=1}^{n} \mathsf{Sol}(\mathcal{D}_i) = \mathsf{Sol}(\mathcal{D})$$

□

Lemma 44
Given $\mathcal{D} \equiv \mathcal{D}' \equiv x \neq a$. *Then* $\mathit{Sol}(\mathcal{D}) \subseteq \mathit{Sol}(\mathcal{D}')$

PROOF. Trivially true. □

We can now use these three lemmas to give a safe approximation to the question "is $\mathsf{Sol}(\mathcal{R}') \subseteq \mathsf{Sol}(\mathcal{R})$?". Let the algorithm *supersetof* be defined below. Given two negative systems $\mathcal{R}, \mathcal{R}'$ in normal form, if $\mathit{supersetof}(\mathcal{R}, \mathcal{R}') = \text{TRUE}$ then $\mathsf{Sol}(\mathcal{R}') \subseteq \mathsf{Sol}(\mathcal{R})$. We use this approximation because we do not know whether $\mathsf{Sol}(\mathcal{R}') \subseteq \mathsf{Sol}(\mathcal{R})$ in general is decidable.

Algorithm 45

$\mathit{onedisjunction}(\mathcal{C}, \mathcal{C}') =$ **let** $\bigvee_{i=1}^{n} \mathcal{D}_i \equiv \mathcal{C}$ **and** $\bigvee_{j=1}^{m} \mathcal{D}'_j \equiv \mathcal{C}'$
in foreach $j \leq m$
if $\nexists i \leq n$ such that $\mathcal{D}'_j \equiv \mathcal{D}_i$ **then return** FALSE
endforeach
return TRUE

$$
\begin{array}{lll}
\mathit{supersetof}(\mathcal{R}, \mathcal{R}') & = & \textbf{if } \mathcal{R} \equiv \top \textbf{ then return } \textsc{true} \\
& & \textbf{else let } \bigwedge_{i=1}^{n} \mathcal{C}_i \equiv \mathcal{R} \textbf{ and } \bigwedge_{j=1}^{m} \mathcal{C}'_j \equiv \mathcal{R}' \\
& & \textbf{in foreach } i \leq n \\
& & \quad \textbf{if } \nexists j \leq m \text{ such that } \mathit{onedisjunction}(\mathcal{C}_i, \mathcal{C}'_j) = \textsc{true} \\
& & \quad \textbf{then return } \textsc{false} \\
& & \quad \textbf{endforeach} \\
& & \textbf{return } \textsc{true}
\end{array}
$$

Example 46
Consider the two negative systems

$$\mathcal{R} \equiv (x \neq \textsc{cons}(z_1, z_2) \vee y \neq z_1 \vee z_3 \neq \textsc{nil}) \wedge (y \neq z_1 \vee z_3 \neq \textsc{nil} \vee x \neq \textsc{nil})$$

$$\mathcal{R}' \equiv (z_3 \neq \textsc{nil} \vee y \neq z_1)$$

Since $\mathit{supersetof}(\mathcal{R}, \mathcal{R}') = \textsc{true}$ we know that $\mathsf{Sol}(\mathcal{R}') \subseteq \mathsf{Sol}(\mathcal{R})$.

4.5 Summary

We have developed an algorithm that can decide whether there exists a solution to a set of constraints of form

$$a_1 = a'_1 \wedge \ldots \wedge a_n = a'_n \wedge a_{n+1} \neq a'_{n+1} \wedge \ldots \wedge a_m \neq a'_m$$

where a ranges over terms that consist of constructors and variables. This algorithm can be used to prune branches from a process tree during program transformation.

We have shown how the positive part of a solution can be extracted as a substitution and we have presented a method for comparing the negative part of a solution to the negative part of another solution. During program transformation, these methods can be used to compare nodes in a process tree to decide whether folding is possible.

Chapter 5

Driving

In this chapter we will be more precise about the notion of process trees introduced in chapter 2, and we will present a driving algorithm that produces process trees of very high quality, *i.e.* they contain no unnecessary tests. As an example, we will apply our algorithm to what has become *the* standard test for measuring the strength of program transformers: the naïve string matcher; and we will show that we indeed pass this test and obtain an efficient Knuth, Morris & Pratt matcher [16].

5.1 Transformation on Trees

As we saw in chapter 2, the process graphs generated by supercompilation are in fact represented as trees where cycles are made implicit by special leaves, namely the ones that are renamings of an ancestor. To make it easier to reason about the trees produced by supercompilation, we will now introduce trees in a formal way following Sørensen [25], which is based on Courcelle [5]. This formalisation leaves out labels on edges which will not be used until chapter 8.

Definition 47
Let $i, j \in \mathbb{N}$, $\delta, \gamma, \mu \in \mathbb{N}^*$ and ϵ denotes the empty string[1]. A *tree* over a set $\mathcal{E}$, denoted $\mathsf{T}(\mathcal{E})$, is a partial map $P : \mathbb{N}^* \rightharpoonup \mathcal{E}$ such that

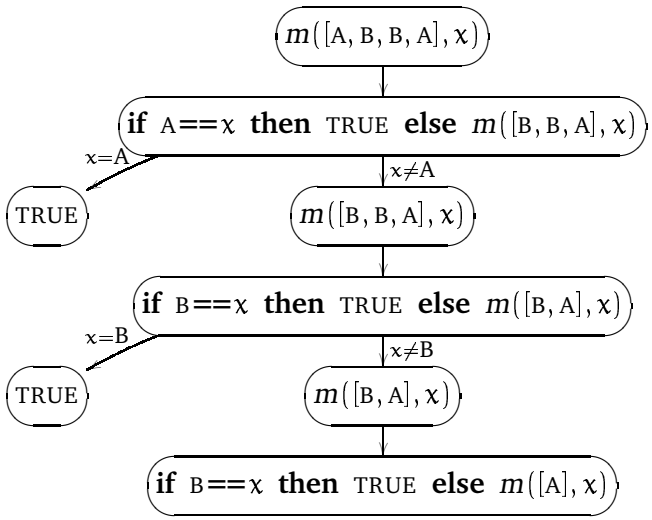
1. P is not empty: $\mathsf{dom}(P) \neq \emptyset$.
2. $\mathsf{dom}(P)$ is prefix-closed: $\delta\gamma \in \mathsf{dom}(P) \Rightarrow \delta \in \mathsf{dom}(P)$.
3. P is finitely branching: $\delta \in \mathsf{dom}(P) \Rightarrow \{i \mid \delta i \in \mathsf{dom}(P)\}$ is finite.
4. P is ordered: $\delta j \in \mathsf{dom}(P) \Rightarrow$ for all $0 \leq i \leq j : \delta i \in \mathsf{dom}(P)$.

The elements of $\mathsf{dom}(P)$ are called *nodes* of P and ϵ is called *the root*. For any node δ in $\mathsf{dom}(P)$, the nodes δi are called the children of δ, and δ is the *parent* of these children. A node with no children is a *leaf*; we denote by $\mathsf{leaf}(P)$ the set of leaves in P. For any node $\delta \in P$, $P(\delta) \in \mathcal{E}$ is the *label* of δ.

A branch in a tree P is a sequence $\delta_0, \delta_1, \delta_2, \ldots \in \mathsf{dom}(P)$ where $\delta_0 = \epsilon$ and $\forall i \in \mathbb{N} : \delta_{i+1}$ is the child of δ_i. If $\mathsf{dom}(P)$ is finite, we say that P is finite; and P is *singleton* if $\mathsf{dom}(P) = \{\epsilon\}$.

[1] As usual in computer science, we define $\mathbb{N} = \{0, 1, 2, \ldots\}$.

Example 48
Let $P \in T(\mathscr{T})$ be a tree from example 3:



P will have domain $\{\epsilon, 0, 00, 01, 010, 0100, 0101, 01010\}$, and *e.g.* $P(01) = m([B, B, A], x)$.

Definition 49
Let $\mathcal{E}$ be a set of symbols, and $P, P' \in T(\mathcal{E})$

1. For $\delta \in \mathsf{dom}(P)$, $P\{\delta := P'\}$ denotes the tree P'':

$$\begin{array}{lcl} \mathsf{dom}(P'') & = & (\mathsf{dom}(P) \setminus \{\delta\gamma \mid \delta\gamma \in \mathsf{dom}(P)\}) \cup \{\delta\gamma \mid \gamma \in \mathsf{dom}(P'')\} \\ P''(\mu) & = & \begin{cases} t'(\gamma) & \text{if } \mu = \delta\gamma \text{ for some } \gamma \\ t(\mu) & \text{otherwise} \end{cases} \end{array}$$

2. Let $\delta \in \mathsf{dom}(P)$. The *ancestors* of δ in P is the set

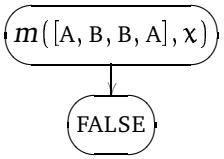
$$\mathsf{anc}(P, \delta) = \{\gamma \in \mathsf{dom}(P) \mid \exists \mu : \delta = \gamma\mu\}$$

   The *proper ancestors* of δ in P is the set

$$\mathsf{propanc}(P, \delta) = \mathsf{anc}(P, \delta) \setminus \{\delta\}$$

3. Let $e \multimap e_1, \ldots, e_n$ denote the tree $P \in T(\mathcal{E})$ with $\mathsf{dom}(P) = \{\epsilon\} \cup \{1, \ldots, n\}$ where $P(\epsilon) = e$ and $P(i) = e_i$. $e \multimap$ thus denotes a leaf.

Example 50
Let $P \in T(\mathscr{T})$ be the tree from example 48. $\mathsf{propanc}(P, 01) = \{\epsilon, 0\}$ and $P\{0 := \text{FALSE} \multimap\} = m([A, B, B, A], x) \multimap \text{FALSE}$, which graphically is the tree



with domain $\{\epsilon, 0\}$.

The previous chapter has provided us with an excellent vehicle for collecting and maintaining information about free variables. We will now incorporate restriction systems into our process tree, to ensure that, in the words of [10], "enough information" is propagated "to be able to prune all infeasible program branches".

5.2 Driving with Restrictions

We can construct the driving algorithm based on the semantics defined in figure 3.3. This is done by extending the small-step semantics to terms that are not closed by speculatively executing tests that depend on variables. For each such speculative execution, information about the tests that have been conducted is collected in a restriction system. As we have seen in the previous chapter, the satisfiability of a restriction system can be calculated, and we can thus at any point use the information collected to decide whether it is feasible to execute a particular branch. The driving algorithm will thus work on pairs of terms and restriction systems, denoted $\langle t, \mathcal{R} \rangle$. To select a component of a pair $\langle t, \mathcal{R} \rangle$ we use the function π_1 or π_2 to select the first or second component, respectively.

Definition 51 (Unfold step)
An *unfold step* is performed by the transition relation $\Rightarrow_{\text{unf}}$ defined by the relations in figure 5.1.

The pattern function application (rule 2) in figure 3.3 has been extended by rule 2b in figure 5.1 that *instantiates* a free variable y to the pattern p taken from the function definition. This is achieved by appending the equation $y = p$ to the current restriction system. If the new restriction system is satisfiable, the function application can be unfolded. In the same manner, rules 4–5 in figure 3.3 have been modified into rules 4–5 in figure 5.1 so that they can handle constructor terms. *i.e.* terms that consist of constructors and variables.

In rule 10, the extraction of a substitution from the restriction system is not strictly necessary, but we will see that it eases the decision of when to fold. The extraction of the substitution also has the effect that it cleans up the restriction system by removing all positive information, including information that has become obsolete (due to disappearance of variables).

When we perform an unfold step, variables can be instantiated. To avoid name capture and other confusion, we must always use fresh variables when we perform an unfold step.

We can now rigorously define how the drive steps shown in chapter 2 should be carried out. Every term in the process tree is accompanied by a restriction system that holds the restrictions on the free variables in the term. We will thus develop trees over $\mathscr{T} \times \mathscr{R}$, where the the root consists of the term we want to drive and the empty restriction system $\top$.

Definition 52 (Drive step)
Let $P \in \mathsf{T}(\mathscr{T} \times \mathscr{R})$ and $\delta \in \mathsf{leaf}(P)$. Then

$$\mathsf{drivestep}(P, \delta) = P\{\delta := P(\delta) \multimap Q_1, \ldots, Q_n\}$$

where $\{Q_1, \ldots, Q_n\} = \{Q \mid P(\delta) \Rightarrow_{\text{unf}} Q\}$.

We asume that the edges in the process tree are annotated by the exact operation that produced the children (as in chapter 2), so the order of children produced by a drive step is not important.

Definition 53 (Instance & Renaming)

1. A term s is an *instance* of term t, denoted $t \preceq s$, if there exists a substitution θ such that $t\theta \equiv s$.

2. A substitution θ is a renaming if it is a bijection from $\mathscr{X}$ to $\mathscr{X}$. A term t is a *renaming of* a term s, denoted $t \doteq s$, if there exists a renaming θ such that $t\theta \equiv s$.

Definition 54 (Final)
Let $P \in \mathsf{T}(\mathscr{T} \times \mathscr{R})$. A node $\delta \in \mathsf{leaf}(P)$ is *final* if one of the following conditions are met:

$\boxed{\langle t, \mathcal{R}\rangle \to \langle t, \mathcal{R}\rangle}$

$$(1)\ \frac{f(x_1, \ldots, x_n)\ =\ s \in q}{\langle f(t_1, \ldots, t_n), \mathcal{R}\rangle \to \langle s\{x_1{:=}t_1, \ldots, x_n{:=}t_n\}, \mathcal{R}\rangle}$$

$$(2a)\ \frac{f(c(x_1, \ldots, x_m), x_{m+1}, \ldots, x_n)\ =\ s \in q}{\langle f(c(t_1, \ldots, t_m), t_{m+1}, \ldots, t_n), \mathcal{R}\rangle \to \langle s\{x_1{:=}t_1, \ldots, x_n{:=}t_n\}, \mathcal{R}\rangle}$$

$$(2b)\ \frac{f(p, x_1, \ldots, x_n)\ =\ s \in q \qquad \mathcal{R}' = \mathcal{R} \wedge [y = p] \qquad \mathsf{satisfiable}(\mathcal{R}')}{\langle f(y, t_1, \ldots, t_n), \mathcal{R}\rangle \to \langle s\{x_1{:=}t_1, \ldots, x_n{:=}t_n\}, \mathcal{R}'\rangle}$$

$$(3)\ \frac{f(p, x_1, \ldots, x_n)\ =\ s \in q \qquad \langle t, \mathcal{R}\rangle \to \langle t', \mathcal{R}'\rangle}{\langle f(t, t_1, \ldots, t_n), \mathcal{R}\rangle \to \langle f(t', t_1, \ldots, t_n), \mathcal{R}'\rangle}$$

$$(4)\ \frac{\mathcal{R}' = \mathcal{R} \wedge [a = a'] \qquad \mathsf{satisfiable}(\mathcal{R}')}{\langle \textbf{if}\ a{==}a'\ \textbf{then}\ s\ \textbf{else}\ s', \mathcal{R}\rangle \to \langle s, \mathcal{R}'\rangle}$$

$$(5)\ \frac{\mathcal{R}' = \mathcal{R} \wedge [a \neq a'] \qquad \mathsf{satisfiable}(\mathcal{R}')}{\langle \textbf{if}\ a{==}a'\ \textbf{then}\ s\ \textbf{else}\ s', \mathcal{R}\rangle \to \langle s', \mathcal{R}'\rangle}$$

$$(6)\ \frac{\langle t_1, \mathcal{R}\rangle \rightsquigarrow \langle t_1', \mathcal{R}'\rangle}{\langle \textbf{if}\ t_1{==}t_2\ \textbf{then}\ s\ \textbf{else}\ s', \mathcal{R}\rangle \to \langle \textbf{if}\ t_1'{==}t_2\ \textbf{then}\ s\ \textbf{else}\ s', \mathcal{R}'\rangle}$$

$$(7)\ \frac{\langle t, \mathcal{R}\rangle \rightsquigarrow \langle t', \mathcal{R}'\rangle}{\langle \textbf{if}\ a{==}t\ \textbf{then}\ s\ \textbf{else}\ s', \mathcal{R}\rangle \to \langle \textbf{if}\ a{==}t'\ \textbf{then}\ s\ \textbf{else}\ s', \mathcal{R}'\rangle}$$

$\boxed{\langle t, \mathcal{R}\rangle \rightsquigarrow \langle t, \mathcal{R}\rangle}$

$$(8)\ \frac{\langle t, \mathcal{R}\rangle \to \langle t', \mathcal{R}'\rangle}{\langle t, \mathcal{R}\rangle \rightsquigarrow \langle t', \mathcal{R}'\rangle}$$

$$(9)\ \frac{\langle t, \mathcal{R}\rangle \rightsquigarrow \langle t', \mathcal{R}'\rangle}{\langle c(a_1, \ldots, a_{i-1}, t, t_{i+1}, \ldots, t_n), \mathcal{R}\rangle \rightsquigarrow \langle c(a_1, \ldots, a_{i-1}, t', t_{i+1}, \ldots, t_n), \mathcal{R}'\rangle}$$

$\boxed{\langle t, \mathcal{R}\rangle \Rightarrow_{\text{unf}} \langle t, \mathcal{R}\rangle}$

$$(10)\ \frac{\langle t, \mathcal{R}\rangle \to \langle t', \mathcal{R}'\rangle \qquad \mathsf{extract}(\mathcal{R}') = (\theta, \mathcal{R}'')}{\langle t, \mathcal{R}\rangle \Rightarrow_{\text{unf}} \langle t'\theta, \mathcal{R}''\rangle}$$

$$(11)\ \frac{}{\langle c(t_1, \ldots, t_n), \mathcal{R}\rangle \Rightarrow_{\text{unf}} \langle t_i, \mathcal{R}\rangle}\ i \in \{1, \ldots, n\}$$

Figure 5.1: Driving with negative information. $n, m \geq 0$.

1. $\mathsf{P}(\delta) = \langle \mathsf{c}(), \mathcal{R} \rangle$, where $\mathsf{c} \in \mathscr{C}$.

2. $\mathsf{P}(\delta) = \langle \mathsf{x}, \mathcal{R} \rangle$, where $\mathsf{x} \in \mathscr{X}$.

3. $\exists \gamma \in \mathsf{propanc}(\mathsf{P}, \delta)$, such that $\mathsf{P}(\gamma) = \langle \mathsf{s}, \mathcal{R}' \rangle$, $\mathsf{P}(\delta) = \langle \mathsf{t}, \mathcal{R} \rangle$ and $\mathsf{s} \doteq \mathsf{t}$ by the renaming θ and $\mathsf{Sol}(\mathcal{R}) \subseteq \mathsf{Sol}(\mathcal{R}'\theta)$.

We say that P is *closed* if all leaves are final.

Algorithm 55
Let $\mathsf{q} \in \mathscr{Q}$ be a program, and define the transformer $\mathsf{M}_\perp : \mathscr{T} \to \mathsf{T}(\mathscr{T} \times \mathscr{R})$ by

input t
$\mathsf{P} = \langle \mathsf{t}, \top \rangle \multimap$
while P is not closed
  let $\delta \in \mathsf{leaf}(\mathsf{P})$ be a non-final node
  $\mathsf{P} = \mathsf{drivestep}(\mathsf{P}, \delta)$
return P

This definition of driving leaves it open which non-final node should be selected in the case that several nodes are available and the definition also postpones code generation concerns. Since this is not the final version of our supercompilation algorithm, we defer these issue to later chapters.

Example 56
Algorithm 55 will generate the process tree depicted in figure 5.2 when given the membership program from example 3 and the term $m([\mathtt{A},\mathtt{B},\mathtt{B},\mathtt{A}], \mathsf{x})$.

Consider again rule 10 in figure 5.1. The extraction of a substitution from the restriction system helps us discover possibilities for folding, since the application of the extracted substitution transfers as much information as possible back into the term. The decision of when to fold can then be based on two conditions: is there an ancestor that is a renaming and, if so, it this ancestor equally or less restricted. The latter can be approximated as we have seen in chapter 4.

Example 57
Consider the silly program

$$\begin{array}{lcl} g(\mathsf{xs}) & = & \textbf{if } \mathsf{xs}{==}[\,] \textbf{ then } \mathsf{xs} \textbf{ else } f(\mathsf{xs}) \\ f([\,]) & = & [\,] \\ f(\mathsf{x} : \mathsf{xs}) & = & f(\mathsf{xs}) \end{array}$$

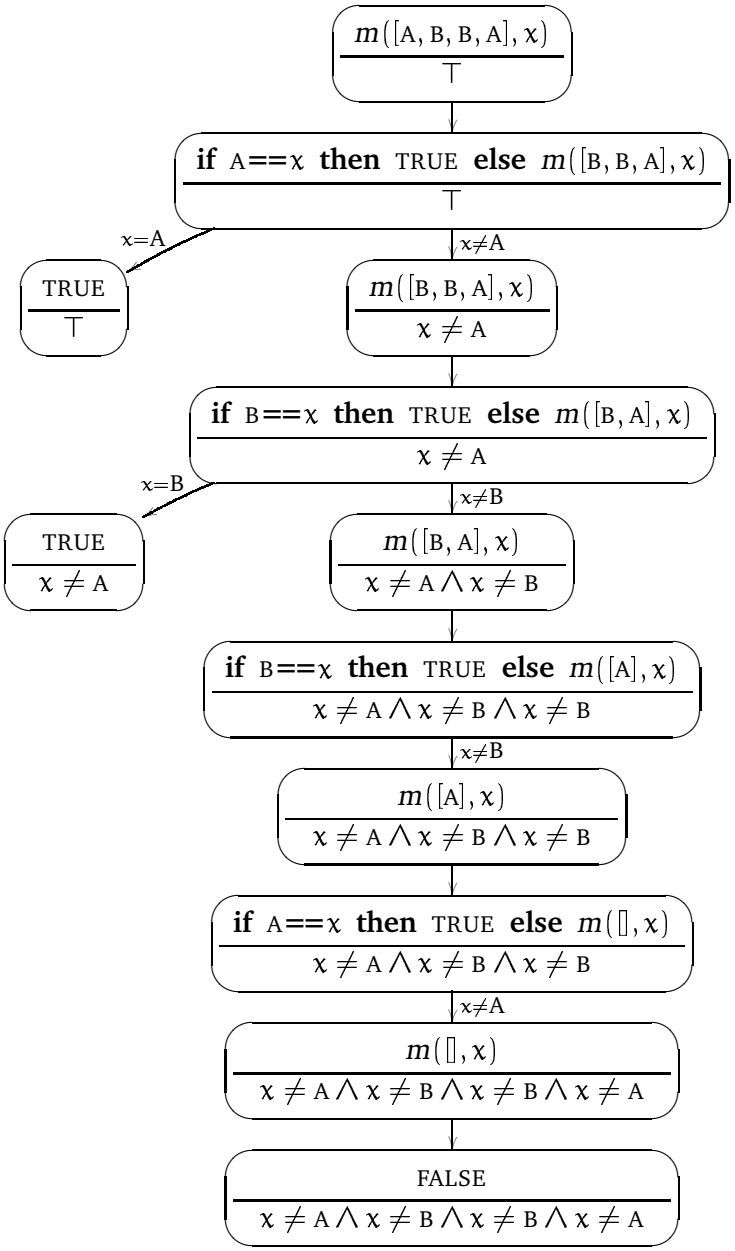
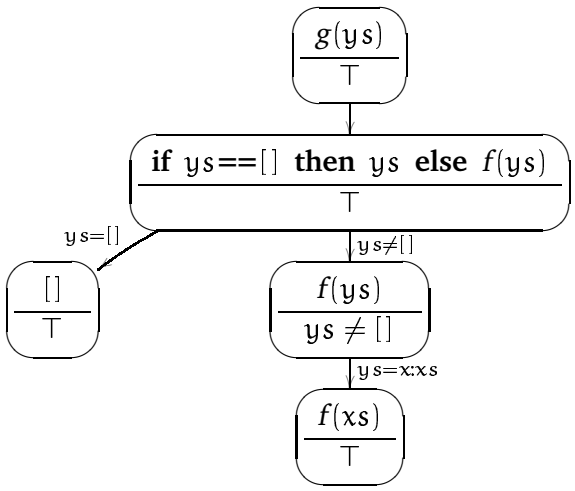
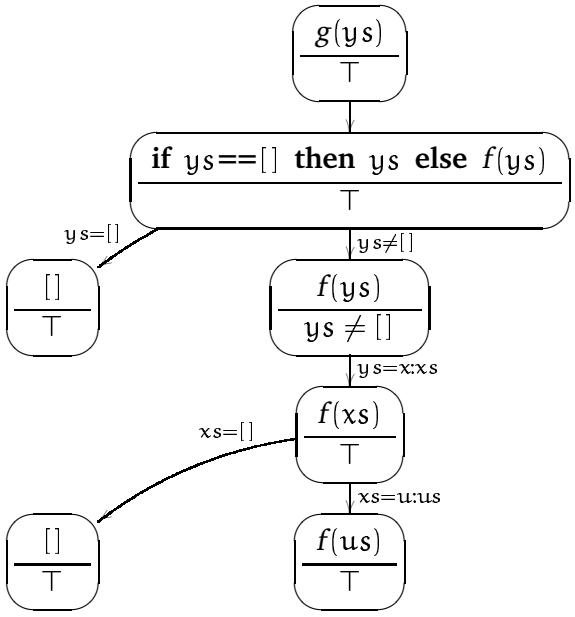


Figure 5.2: Process tree for the membership test in the fixed list $[A, B, B, A]$.

if we drive the term $g(ys)$, the following process tree will be developed:



The right-most leaf node cannot be folded back to its parent, since the parent is more restricted. We will therefore have to continue unfolding:



Let us end this section by relating our unfold rules to other program transformation techniques.

1. If we removed the propagation of negative information in rule 5, we would achieve positive supercompilation [27].

2. If we further removed the restriction systems and instead applied the substitution $\{y := p\}$ directly to function body in rule 2b and applied the substitution $\mathrm{mgu}(a, a')$ to s in rule 4, we would achieve deforestation [37].

3. If we removed the restriction systems altogether, we would achieve something similar to partial evaluation [13].

5.3 Process Graphs

In an attempt to establish a formalised framework in which comparisons of program specializers can be conducted, Glück & Klimov presented in [10] the notion of *perfect process graphs* which stems from [30].

A process graph will be a model of the computations performed by some program q with respect to some initial configuration c_0.

Definition 58 (Process graph)
Let $s \in S$ range over states, $c \in \mathscr{P}(S)$ range over configurations (sets of states), $p \subseteq \mathscr{P}(S)$ range over predicates on configurations and let $\Rightarrow \subseteq S \times S$ be a transition relation on states, and let $\Rightarrow^*$ denote the transitive, reflexive closure of this relation.

A *process graph* is a rooted, directed graph where each node is labelled by some configuration c, and each edge (c, c') represents a transition from a state in c to a state in c', and is labelled by a predicate p, denoted $c \multimap_p c'$. Let the root be denoted c_0, which is called the *initial configuration*. A state that is contained in the initial configuration is called an *initial state*.

Definition 59 (Correct process graph)
A process graph is *correct* if, for any node c in the process graph

1. for all $s \in c$, if there exists s', s_0 such that $s_0 \in c_0$, $s_0 \Rightarrow^* s$ and $s \Rightarrow s'$, then there exists p, c' such that $s' \in c'$ and $c \multimap_p c'$; and
2. for all edges $c \multimap_{p_i} c'_i$, the predicates $p_1, \ldots, p_n$ divide c into disjoint subsets.

The intuitive interpretation of the above definition is that a configuration that branches to several other configurations will represent a conditional transition. The edges on such branching configurations are labelled with a test on the configuration, which signifies that, if the test is successful, control can be transfered to that configuration.

Definition 60 (walk)
A *walk* in a process graph P is a sequence of nodes and edges through P.

By the first condition in definition 59, any specific computation will have at least one walk in the process graph, and by the second condition it will have at most one walk. Hence, any specific computation will follow a unique walk in the process graph.

We can now introduce a notion of quality of process graphs: if a process graph (for some program and an initial configuration) contains a configuration that will never be reached by any computation, the process graph cannot be considered an optimal model of the computations performed.

Definition 61 (Perfect process graph)
Given some process graph P,

1. a walk in P is *feasible* if there exists an initial state such that computation follows this walk.
2. a node in P is *feasible* if it belongs to some feasible walk.
3. P is *perfect* if all walks are feasible.

The ultimate goal of our program transformer is to produce perfect process graphs – *i.e.* graphs that are free from infeasible nodes – since infeasible nodes reduces the efficiency of the computation process via redundant tests. To see this, consider an infeasible configuration c.

Since the root of the process graph represents the initial configuration, it will be contained in any computation and is thus feasible. It must then be the case that there is a configuration c' on an infeasible walk from the root to c such that c' is the last feasible configuration in this walk. Since c' is feasible, some feasible walk must go through c'. Since also an infeasible walk goes through c', c' is a branching node. Hence the test on c' that transfers control to the infeasible walk is not needed since it can never be satisfied by any computation sequence.

We will now see how process trees can be regarded as process graphs, and thereby assess the quality of our program transformer. Since our language evaluates closed terms, our notion of states and configurations is somewhat different: let closed terms constitute the states, pairs of terms and restriction systems constitute the configurations, and let the transition relation be $\Rightarrow_{\text{cbn}}$. A closed term s is *contained* in a configuration $\langle t, \mathcal{R}\rangle$ if and only if there exists a value substitution $\underline{\theta}$ such that $s \equiv t\underline{\theta}$ and $\mathcal{R}$ is trivially satisfied by $\underline{\theta}$. Lastly, let a leaf that is a renaming of some ancestor represent a cycle in the graph.

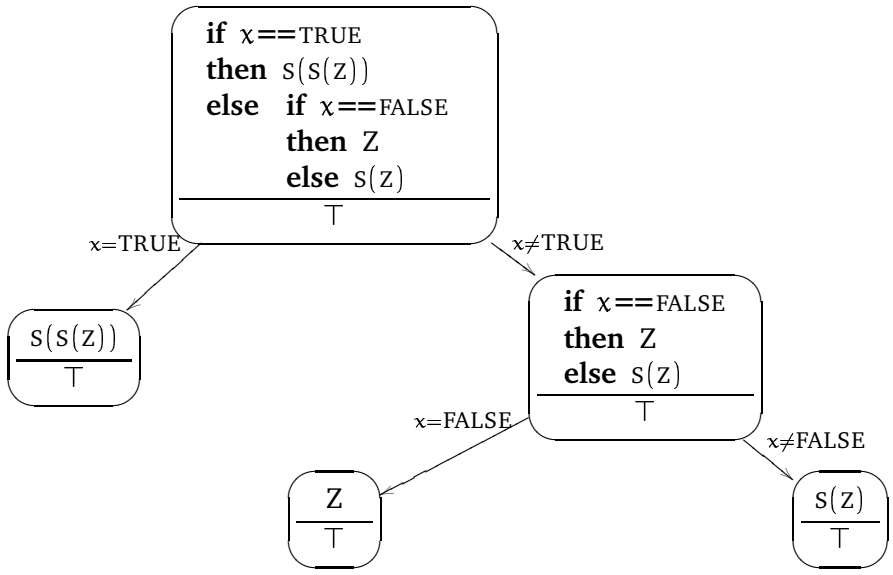
Example 62
Consider the "program"

$$\begin{array}{lcl} \textbf{data}\ \text{Bool}() & = & \text{TRUE} \mid \text{FALSE} \\ \textbf{data}\ \text{Num}() & = & \text{S}(\text{Num}()) \mid \text{Z} \end{array}$$

and the initial configuration

$$\langle \textbf{if}\ x{==}\text{TRUE}\ \textbf{then}\ \text{S}(\text{S}(\text{Z}))\ \textbf{else}\ \textbf{if}\ x{==}\text{FALSE}\ \textbf{then}\ \text{Z}\ \textbf{else}\ \text{S}(\text{Z}), \top\rangle$$

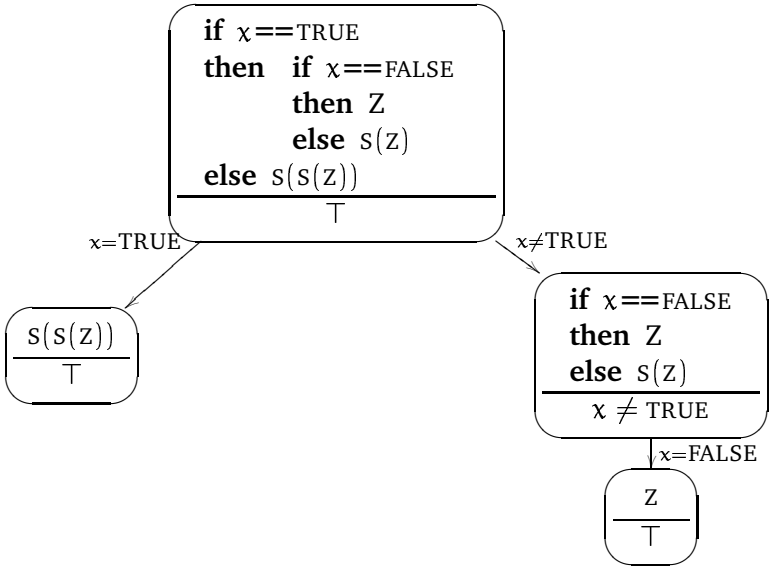
A hypothetical but correct process graph (in this case, a tree) for this configuration could be



The right-most branch can never be reached by any computation, and it is thus infeasible. Hence the process graph is imperfect.

If we now use our driving algorithm on the same program and initial configuration, the

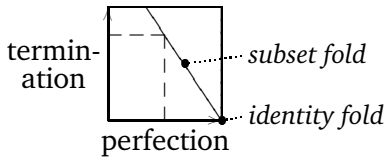
following perfect process graph will be produced:



From this process graph we can reconstruct the new and improved term

if x==TRUE **then** S(S(Z)) **else** Z

We would very much suspect that our driving algorithm produces correct process graphs, since each driving step closely models the semantics of our language. The proof for this, however, is very complex and has been omitted from this text since this property seems fairly obvious. A more interesting question is whether the process graphs constructed are perfect. The answer is no, since we allow the algorithm to fold back to a node that represents a more general configuration.



The core of our algorithm – the drive step – is, however, perfect: if we change our algorithm so that it only folds on identical configurations, we would indeed produce perfect process graphs but we would then also have increased the danger of non-termination. This tradeoff is depicted in the graph above. We shall later see that the perfectness of the process graphs will degrade even further as we strive to ensure termination.

5.4 Local unfolding

If we look at the process tree in figure 5.2, we will see that some parts of the tree are created by *deterministic unfolding*, *i.e.* they each consists of a single path. This is a good sign, since it means that they represent local computations that will *always* be carried out when the program is in this particular configuration, regardless of the unknown input. We can thus do these intermediate computations – or precomputations as they are called in partial evaluation – once and for all, throw away the intermediate steps and just remember the results. Such *local unfoldings* will not only decrease the size of our process tress, they will also allow us to ensure that folding is not carried out prematurely.

Inspired by [19], we will now formulate an improved driving mechanism that does not keep the intermediate paths and only put nodes in the process tree if they represent computations that are needed in the transformed program. This will furthermore have desirable effects when the transformed program is generated, as we shall see in chapter 6.

Algorithm 63
Let $q \in \mathscr{Q}$ be a program, and define $M_{\perp G} : \mathscr{T} \to T(\mathscr{T} \times \mathscr{R})$ by

input t
$P = \langle t, \top \rangle \multimap$
while P is not closed
  let $\delta \in \mathsf{leaf}(P)$ be a non-final node
  $Q = P(\delta) \multimap$
  do
    let $\gamma \in \mathsf{leaf}(Q)$
    $Q = \mathsf{drivestep}(Q, \gamma)$
  until $|\mathsf{leaf}(Q)| > 1$ or $Q(\gamma) = \langle c(t_1, \ldots, t_n), \mathcal{R} \rangle$
      or ($\mu \in \mathsf{leaf}(Q)$ and ($\mathsf{Var}(Q(\mu)) \not\subseteq \mathsf{Var}(Q(\gamma))$ or μ is final))
  let $\mu \in \mathsf{leaf}(Q)$
  if μ is final
    $P = P\{\delta := Q(\mu) \multimap\}$
  else
    $P = P\{\delta := Q(\gamma) \multimap Q(\gamma 0), Q(\gamma 1), \ldots\}$
return P

The algorithm works as follows. A *global* process tree P is created with the initial configuration as root. When a leaf in P is selected to be unfolded, a *local* process tree Q is created such that it has the selected node as root. Q is now unfolded until either

1. unfolding results in more than one child, or
2. a term with an outermost constructor has been unfolded, or
3. an instantiation of variables has taken place, or
4. the leaf is final.

In the first case, the outcome of a test could not be decided, so this test must be represented in the transformed program. In the second case, a term with an outermost constructor has been met, so this data construction must be present in the transformed program. In the third case, the outcome of a pattern match could be decided, but since a variable has been instantiated, a pattern match must be present in the transformed program. In these three cases, the node that resulted in the test (or construction) and its children is put back into the global tree. In the fourth case, no further unfolding can be done or it is possible to fold, so the leaf is put back into the global tree.

Example 64
Consider again the string-matcher program from example 5.

$$
\begin{array}{lcl}
\textbf{data}\ \mathrm{Bool}() & = & \textsc{true} \mid \textsc{false} \\
\textbf{data}\ \mathrm{Letter}() & = & \textsc{a} \mid \textsc{b} \mid \textsc{c} \mid \ldots \\
\textbf{data}\ \mathrm{List}(a) & = & \textsc{cons}(a, \mathrm{List}(a)) \mid \textsc{nil} \\
 & & \\
\mathit{match}(p, s) & = & \mathit{m1}(p, s, p, s) \\
 & & \\
\mathit{m1}(\textsc{nil}, s, op, os) & = & \textsc{true} \\
\mathit{m1}(\textsc{cons}(p, pp), s, op, os) & = & \mathit{m2}(s, p, pp, op, os) \\
 & & \\
\mathit{m2}(\textsc{nil}, p, pp, op, os) & = & \textsc{false} \\
\mathit{m2}(\textsc{cons}(s, ss), p, pp, op, os) & = & \textbf{if}\ s\texttt{==}p\ \textbf{then}\ \mathit{m1}(pp, ss, op, os)\ \textbf{else}\ \mathit{next}(os, op) \\
 & & \\
\mathit{next}(\textsc{cons}(s, ss), p) & = & \mathit{m1}(p, ss, p, ss)
\end{array}
$$

Given the initial term $\mathit{match}([\textsc{a},\textsc{a},\textsc{b}], zs)$, algorithm 63 will generate the global process graph in figure 5.3.

The intermediate local process trees are depicted in figure 5.4. Notice how local tree number 2 eliminates the test "$\textsc{a} = x$?" by use of negative information.

The resulting program is shown in figure 5.5. It is optimal in the sense that it only traverses the input string once and does not perform any unnecessary tests. The general string matcher algorithm has thus been transformed into the finite-state machine depicted in figure 5.6.

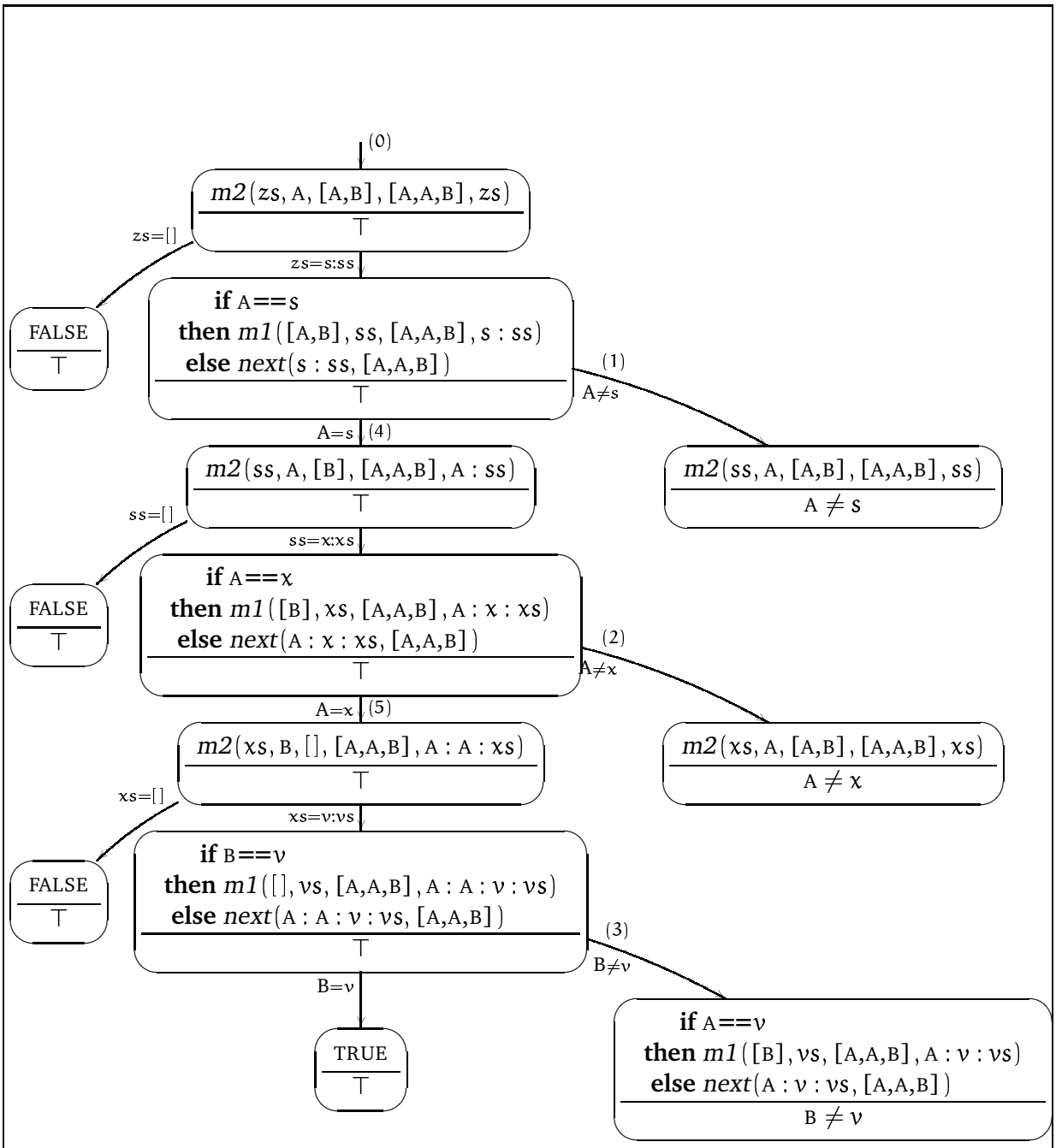


Figure 5.3: Global process tree for the string matcher. Left-out local trees are indicated by numbers 0–5.

5.5 Summary

The naïve string-matcher has been used to measure the strength of various program transformers, and we have shown that our driving algorithm passes test by producing an optimal KMP-matcher. But there is a downside: the driving algorithm does not terminate on a wide class of programs. This is not surprising, since it is not, in general, possible to construct a finite, perfect process graph.

This fact is not reason for despair; to quote from [10]: "Once a perfect driving mechanism is constructed, it is solid ground for the further development. As a result, the problem of approximation has been driven into one corner: folding." In the next chapter we will present a solution to the termination problem.

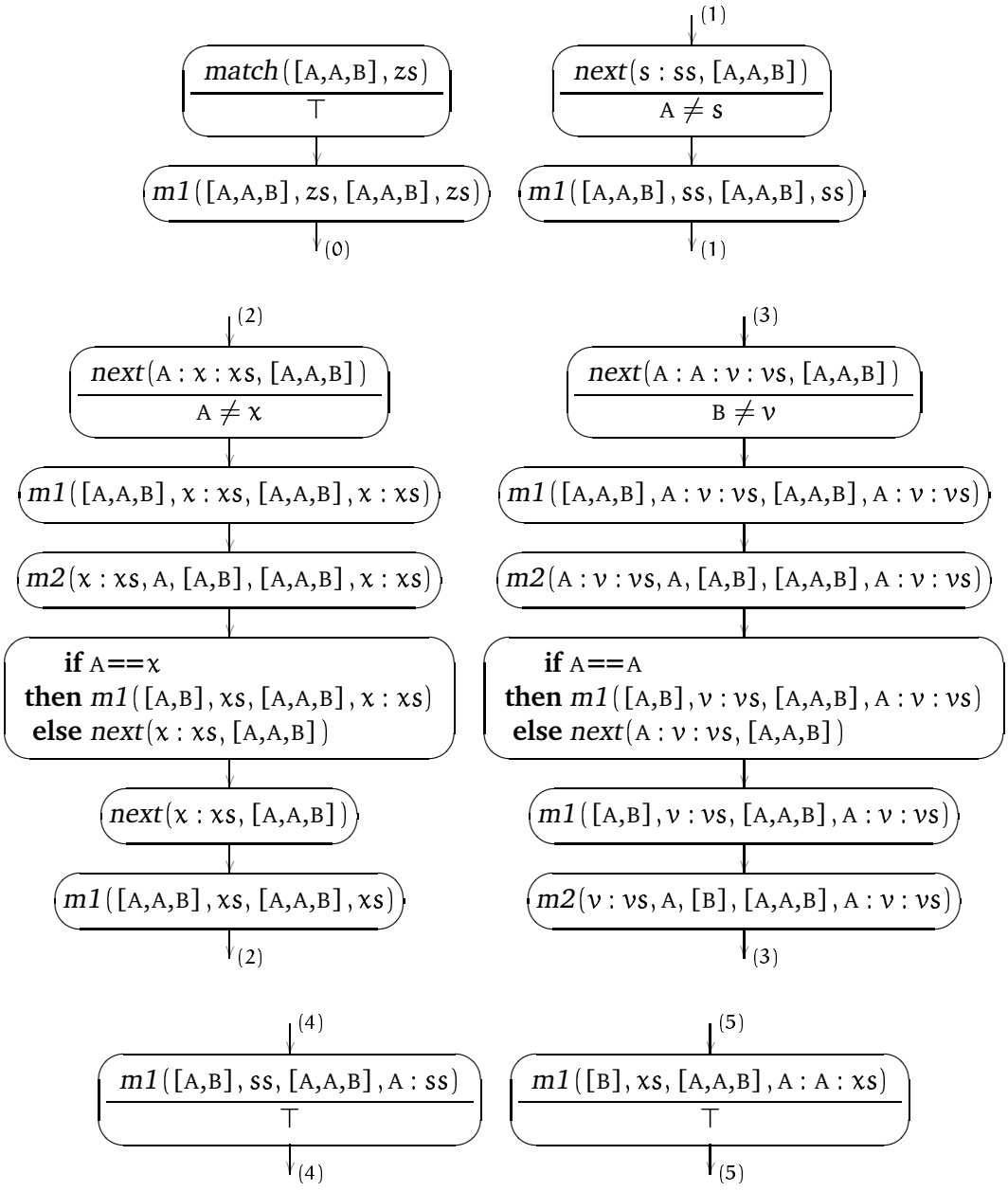


Figure 5.4: Local process trees for the string matcher.

$$
\begin{array}{lcl}
\mathit{m2\_aab}([\,]) & = & \text{FALSE} \\
\mathit{m2\_aab}(x : xs) & = & \mathit{f\_aab}(x, xs) \\
\mathit{f\_aab}(x, xs) & = & \textbf{if } \text{A}{==}x \textbf{ then } \mathit{m2\_ab}(xs) \textbf{ else } \mathit{m2\_aab}(xs) \\
\mathit{m2\_ab}([\,]) & = & \text{FALSE} \\
\mathit{m2\_ab}(x : xs) & = & \mathit{f\_ab}(x, xs) \\
\mathit{f\_ab}(x, xs) & = & \textbf{if } \text{A}{==}x \textbf{ then } \mathit{m2\_b}(xs) \textbf{ else } \mathit{m2\_aab}(xs) \\
\mathit{m2\_b}([\,]) & = & \text{FALSE} \\
\mathit{m2\_b}(x : xs) & = & \mathit{f\_b}(x, xs) \\
\mathit{f\_b}(x, xs) & = & \textbf{if } \text{B}{==}x \textbf{ then } \text{TRUE} \textbf{ else } \mathit{f\_ab}(x, xs)
\end{array}
$$

Figure 5.5: Supercompiling a naïve string matcher with fixed pattern [A,A,B] results in a KMP matcher.

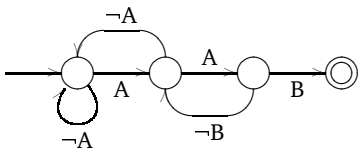


Figure 5.6: The transformed program as a finite-state machine.

Chapter 6

Generalisation

As suggested in the previous chapters, driving does not always terminate because infinite process trees can be produced: the simple "fold when possible" strategy does not ensure termination. In this chapter we will present a full supercompilation algorithm that ensures termination by means of generalisations.

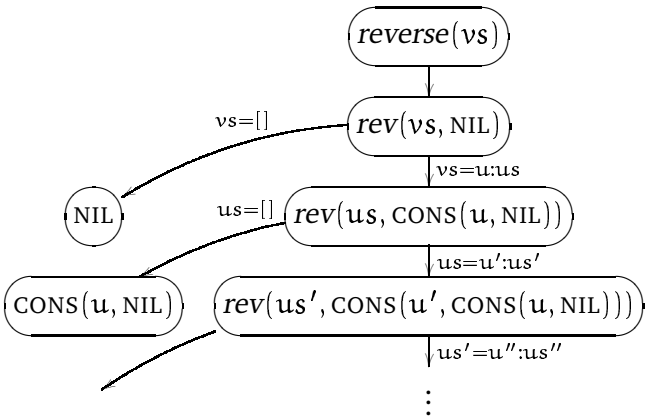
6.1 Characterisation of Non-termination

It turns out that the reasons for non-termination in positive supercompilation can be narrowed down to three canonical patterns, as described in [24]. When negative information is involved, a fourth pattern occurs. We will now illustrate these patterns with four examples. The first three are taken from [24] and in these examples we will omit the restriction systems, since no negative information is propagated.

Example 65 (Accumulating Parameter)
Consider this program that reverses a list:

$$\begin{array}{lcl} \textbf{data}\ \mathrm{List}(a) & = & \mathrm{CONS}(a, \mathrm{List}(a)) \mid \mathrm{NIL} \\ \mathit{reverse}(xs) & = & \mathit{rev}(xs, \mathrm{NIL}) \\ \mathit{rev}(\mathrm{NIL}, ys) & = & ys \\ \mathit{rev}(\mathrm{CONS}(x, xs), ys) & = & \mathit{rev}(xs, \mathrm{CONS}(x, ys)) \end{array}$$

Driving the term $\mathit{reverse}(vs)$ will lead to

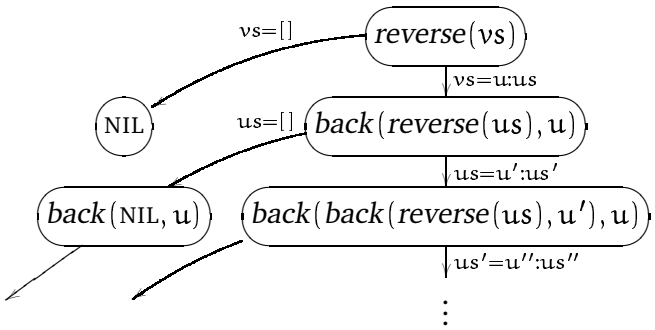


The reason for non-termination is that the accumulating parameter keeps growing, which makes folding impossible.

Example 66 (Obstructing Function Call)
Consider another formulation of a program that reverses a list:

$$\begin{array}{lcl}
\textbf{data}\ \mathrm{List}(a) & = & \mathrm{CONS}(a, \mathrm{List}(a)) \mid \mathrm{NIL} \\
\mathit{reverse}(\mathrm{NIL}) & = & \mathrm{NIL} \\
\mathit{reverse}(\mathrm{CONS}(x, xs)) & = & \mathit{back}(\mathit{reverse}(xs), x) \\
\mathit{back}(\mathrm{NIL}, y) & = & \mathrm{CONS}(y, \mathrm{NIL}) \\
\mathit{back}(\mathrm{CONS}(x, xs), y) & = & \mathrm{CONS}(x, \mathit{back}(xs, y))
\end{array}$$

Driving the term $\mathit{reverse}(vs)$ will lead to

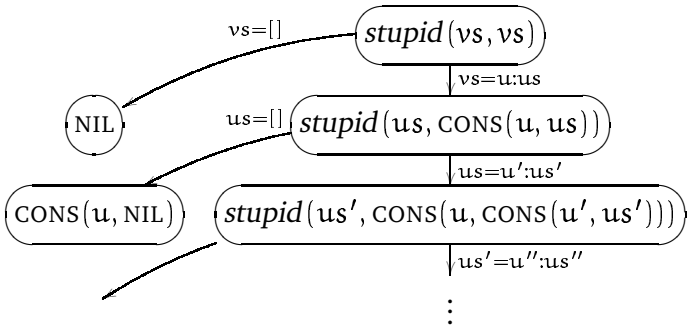


The reason for non-termination is that the inner call obstructs evaluation calls such that increasingly many function calls are put in suspension. Folding is thus impossible.

Example 67 (Accumulating Side-effect)
Consider now the following strange program:

$$\begin{array}{lcl}
\textbf{data}\ \mathrm{List}(a) & = & \mathrm{CONS}(a, \mathrm{List}(a)) \mid \mathrm{NIL} \\
\mathit{stupid}(\mathrm{NIL}, ys) & = & ys \\
\mathit{stupid}(\mathrm{CONS}(x, xs), ys) & = & \mathit{stupid}(xs, ys)
\end{array}$$

Driving the term $\mathit{stupid}(vs, vs)$ will lead to

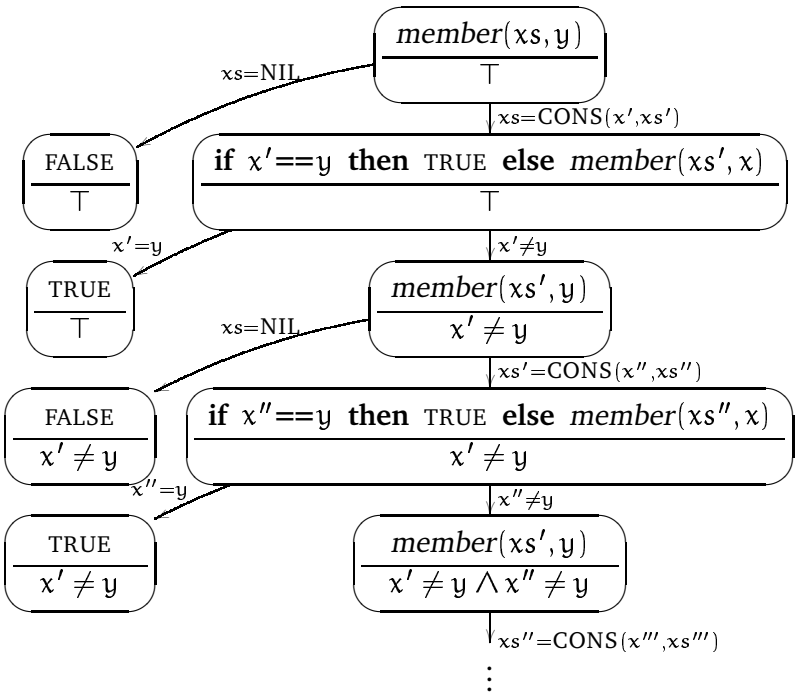


The problem here is that the aggressive information propagation. The second argument in the function call is ever growing as a side-effect of the instantiations of the first argument. Again, folding is not possible.

Example 68 (Accumulating Negative Information)
Consider again the membership program from chapter 2.

$$
\begin{array}{lcl}
\textbf{data}\ \mathrm{List}(a) & = & \textsc{cons}(a, \mathrm{List}(a)) \mid \textsc{nil} \\
\mathit{member}(\textsc{nil}, v) & = & \textsc{false} \\
\mathit{member}(\textsc{cons}(u, us), v) & = & \textbf{if}\ u{=}{=}v\ \textbf{then}\ \textsc{true}\ \textbf{else}\ \mathit{member}(us, v)
\end{array}
$$

Driving the term $\mathit{member}(xs, y)$ will lead to



if we only fold when two nodes are the same modulo renaming.

The first two, The Accumulating Parameter, and The Obstructing Function Call are known problems from deforestation. The Accumulating Side-effect is special for supercompilation due to the additional strength offered by driving. Accumulating Negative Information is special for supercompilers propagating negative information.

6.2 When to stop

By inspection of these examples, it is clear that the non-termination is somehow connected with unbounded growth in the nodes in the process tree. If the nodes do not grow, we can handle infinite reduction sequences by keeping a record of all previous terms, as we have seen in the previous chapter. Accumulating negative information can be handled by folding when a configuration is more restricted than a previously encountered configuration. We have found no method for deciding this general. In cases where we cannot decide whether a configuration c is more restricted than a previously encountered configuration c', we will simply discard the negative information in c' and restart driving from c', which will ensure that the problem does not occur again.

This still leaves us with the problem of ever growing terms. So how can we recognise a potential infinite reduction sequence? The immediate answer is to set some bound on the size of terms, but this is not a very satisfactory solution. Luckily, Sørensen [24] deduced from the previous examples that the core of the problem with the growing terms lies in the way each term in such infinite sequence seems to embed itself in the following term in an intricate way. He then found the *homeomorphic embedding* relation known from static termination analysis in term rewrite system theory, and adapted it to our setting were there are an unbounded number of variables. For a full treatment of the relation, see [8].

Definition 69
Let $\mathscr{H} = \mathscr{F} \cup \mathscr{C} \cup \{\textbf{ifthenelse}\}$. The *homeomorphic embedding* $\trianglelefteq$ is the smallest relation on $T(\mathscr{H} \cup \mathscr{X})$ such that, for all $h \in \mathscr{H}$, $x, y \in \mathscr{X}$, and $t_i, t_i' \in T(\mathscr{H} \cup \mathscr{X})$:

$$\frac{}{x \trianglelefteq y} \qquad \frac{\exists i \in \{1, \ldots, n\} : t \trianglelefteq t_i'}{t \trianglelefteq h(t_1', \ldots, t_n')} \qquad \frac{\forall i \in \{1, \ldots, n\} : t_i \trianglelefteq t_i'}{h(t_1, \ldots, t_n) \trianglelefteq h(t_1', \ldots, t_n')}$$

Example 70
Let $x, y \in \mathscr{X}$, and $b, c, d, f \in \mathscr{H}$. The following terms give examples and non-examples of homeomorphic embedding:

$$\begin{array}{rcl@{\qquad}rcl} b & \trianglelefteq & f(b) & f(c(b)) & \not\trianglelefteq & c(b) \\ c(b) & \trianglelefteq & c(f(b)) & f(c(b)) & \not\trianglelefteq & c(f(b)) \\ d(b,b) & \trianglelefteq & d(f(b), f(b)) & f(c(b)) & \not\trianglelefteq & f(f(f(b))) \\ d(x,y) & \trianglelefteq & d(f(y), x) & d(x,y) & \not\trianglelefteq & d(f(y), f(b)) \end{array}$$

If we return to the first three examples of non-terminating programs, we can by close re-examination see that in all three infinite reduction sequences, every term is homeomorphically embedded in the succeeding term. In fact, due to Higman and Kruskal, the following holds:

Theorem 71 (Kruskal's Tree Theorem)
Let $t_0, t_1, \ldots$ be an infinite sequence of terms over a finite set of constructor names, function names and variables. Then there exists $i < j$ such that $t_i \trianglelefteq t_j$.

The homeomorphic embedding then qualifies as a stop criterion: in every infinite sequence there will be two terms such that the latter embeds the other. Conversely, if at leaf t' we have that $t \trianglelefteq t'$, it will be the case that every subterm of t is embedded in t', which seems to indicate that no real progress was made by this branch; driving is thus stopped for a good reason.

Armed with the homeomorphic embedding relation as a means to stop potential infinite branches we can now modify the core of the driving algorithm from section 5.4 so that it stops when there is a danger of non-termination. The algorithm takes as input a process tree, which

we will call the global tree, and a node in this tree. It then creates a local tree consisting of this node only and unfolds this intermediate tree until either

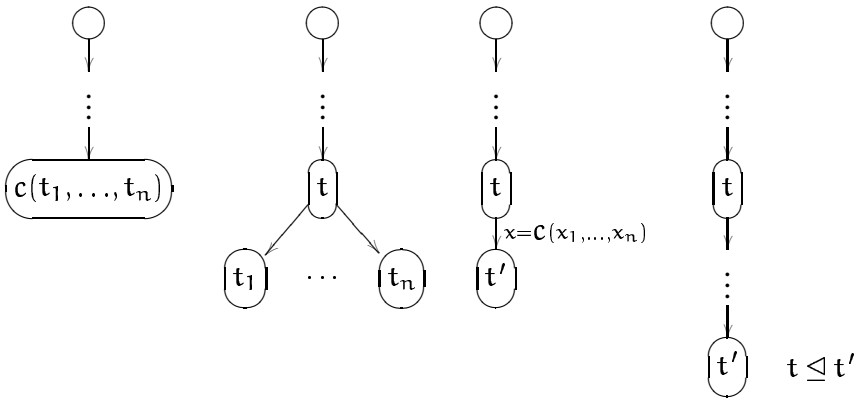
1. no more unfoldings are possible, or
2. an outermost constructor is met, or
3. the tree branches, or
4. an instantiation of variables has taken place, or
5. the newly added leaf embeds another node in the tree.

The "interesting" nodes are then extracted from the local tree and added to the global tree. The algorithm is shown below, and it will be the main engine in our supercompiler.

Algorithm 72
Define $\mathsf{drive} : \mathsf{T}(\mathscr{T} \times \mathscr{R}) \times \mathbb{N}^* \to \mathsf{T}(\mathscr{T} \times \mathscr{R})$ by

input P, δ
let $Q = P(\delta) \multimap$
repeat
  let $\gamma \in \mathsf{leaf}(Q)$ and $\mu = \gamma$
  $Q = \mathsf{drivestep}(Q, \gamma)$
  let $\gamma \in \mathsf{leaf}(Q)$ and $Q(\gamma) = \langle t, \mathcal{R} \rangle$
until $t \in \mathscr{X}$ or $t = c(t_1, \ldots, t_n)$ or $|\mathsf{leaf}(Q)| > 1$ or
    $\mathsf{Var}(Q(\gamma)) \not\subseteq \mathsf{Var}(Q(\mu))$ or $\exists \delta' \in \mathsf{propanc}(Q, \gamma)$ s.t. $\pi_1(Q(\delta')) \trianglelefteq t$
if $t \in \mathscr{X}$ or $t = c(t_1, \ldots, t_n)$
  $P = P\{\delta := Q(\gamma) \multimap\}$
else if $\exists \delta' \in \mathsf{propanc}(Q, \gamma) : \pi_1(Q(\delta')) \trianglelefteq t$
  $P = P\{\delta := Q(\delta') \multimap Q(\gamma)\}$
else
  $P = P\{\delta := Q(\mu) \multimap Q(\mu 0), Q(\mu 1), \ldots\}$
return P

The different developments of the local process tree can be depicted thus:



The labelled nodes are those of interest. When driving stops, the nodes of interest are added to the global process tree.

6.3 How to Stop

Since we want our supercompiler to produce a program, we cannot just throw our hands in the air and give up when driving seems to be non-terminating. As we saw in chapter 2, there is a way to gently put a foot on the brake, namely by making generalisations. The purpose of generalisation in supercompilation is to loosen up a term, so to speak, so that the components can be treated separately and driving can be continued. For this purpose we will introduce let-terms.

Definition 73 (Let-terms)
Let $t \in \mathscr{T}$ range over terms. The set of *let-terms* is defined as follows:

$$\begin{array}{rcl} \mathscr{L} \ni v & ::= & t \\ & \mid & \textbf{let } x_1 = t_1, \ldots, x_n = t_n \textbf{ in } t_0 \end{array}$$

where $n > 0$ and

1. $t_0 \notin \mathscr{X}$,
2. $x_1, \ldots, x_n \in \mathsf{Var}(t_0)$,
3. $t_0\{x_1 := t_1, \ldots, x_n := t_n\} \neq t_0$, and
4. $\forall i, j : i \neq j$ implies $x_i \neq x_j$.

A let-term is considered *proper* if it belongs to $\mathscr{L} \setminus \mathscr{T}$. A let-term in $\mathscr{T}$ is called a *regular* term.

Examples of let-terms	Non-examples of let-terms
$\textbf{let } x = y \textbf{ in } P(y, x)$	$\textbf{let } x = y \textbf{ in } P(x, x)$
$\textbf{let } x = \text{NIL}, y = P(z, \text{NIL}) \textbf{ in } f(x, y)$	$\textbf{let } x = \text{NIL}, y = \text{NIL} \textbf{ in } f(x)$

Table 6.1: Examples of let-terms

Notice that a regular term always can constructed from a let-term by substituting the let-bound terms on the left into the term on the right. Let-terms can thus be regarded simply as a means to decompose a regular term and is thus not an extension of the syntax of our object language.

We will now see how regular terms can be decomposed into let-terms via generalisations. If generalisation of some term is invoked, it will always be because some other term in the process tree endangers termination. When we make a generalisation from two terms, we want the generalisation to be as specific as possible, *i.e.* we want to preserve as much common structure as possible. The following definition helps us to achieve exactly that.

Definition 74 (Generalisation)

1. A *generalisation* of two terms t, s is a term u such that $u \preceq t$ and $u \preceq s$.
2. A *most specific generalisation* (msg) of two terms t, s is a generalisation u such that, for all generalisation u' of t, s, $u \preceq u'$.

Proposition 75 (Unique MSG)
Let $t, s \in \mathscr{T}$ be terms. There exists exactly one msg of t, s modulo renaming.

See Lassez, Maher & Marriott [17] for a proof. The most specific generalisation and the corresponding substitutions can be obtained by exhaustive application of the following rewrite system to the initial tuple $(x, \{x := t\}, \{x := t'\})$, where $t, t' \in \mathscr{T}$, $x, y \in \mathscr{X}$ and $h \in \mathscr{F} \cup \mathscr{C} \cup \{\textbf{ifthenelse}\}$:

$$\begin{pmatrix} t_g \\ \{x{:=}h(t_1,\ldots,t_n)\} \cup \theta_1 \\ \{x{:=}h(t'_1,\ldots,t'_n)\} \cup \theta_2 \end{pmatrix} \rightarrow \begin{pmatrix} t_g\{x{:=}h(y_1,\ldots,y_n)\} \\ \{y_1{:=}t_1,\ldots,y_n{:=}t_n\} \cup \theta_1 \\ \{y_1{:=}t'_1,\ldots,y_n{:=}t'_n\} \cup \theta_2 \end{pmatrix}$$

$$\begin{pmatrix} t_g \\ \{x{:=}t, y{:=}t\} \cup \theta_1 \\ \{x{:=}t', y{:=}t'\} \cup \theta_2 \end{pmatrix} \rightarrow \begin{pmatrix} t_g\{x{:=}y\} \\ \{y{:=}t\} \cup \theta_1 \\ \{y{:=}t'\} \cup \theta_2 \end{pmatrix}$$

Definition 76 ($\sqcap$ and $\leftrightarrow$)

1. Given two terms t, s, the operation $t \sqcap s$ results in a triple (u, θ_1, θ_2) where u is the msg of t, s and $t = u\theta_1$ and $s = u\theta_2$.
2. Two terms t, s are *incommensurable*, denoted $t \leftrightarrow s$, if $t \sqcap s = (x, \theta_1, \theta_2)$.

t_1	t_2	$t_1 \sqcap t_2$
c	$f(c)$	$(x, \{x{:=}c\}, \{x{:=}f(c)\})$
$c(x)$	$c(f(y))$	$(c(x), \{\}, \{x{:=}f(y)\})$
$f(c,c)$	$f(g(c), g(c))$	$(f(x,x), \{x{:=}c\}, \{x{:=}g(c)\})$

Table 6.2: Examples of generalisations.

Based on the unique most specific generalisation of two terms, we can create a let-term that maintains the common structure. If the most specific generalisation of two terms is a variable, however, there will be no such common structure that can be expressed in the form of a let-term, and special measures have to be taken. Inspired by [26], we can define two generalisation operations on nodes in the process tree based on the form of the msg.

Definition 77
Let $P \in \mathsf{T}(\mathscr{L} \times \mathscr{R})$.

1. For $\gamma, \delta \in \mathsf{dom}(P)$ with $P(\gamma) = \langle t, \mathcal{R}\rangle$, $P(\delta) = \langle t', \mathcal{R}'\rangle$ where
   (a) $t, t' \in \mathscr{T}$, and
   (b) $t \sqcap t' = (s, \{x_1{:=}s_1, \ldots, x_n{:=}s_n\}, \theta)$, and
   (c) $s \notin \mathscr{X}$, and
   (d) $s\{x_1{:=}s_1, \ldots, x_n{:=}s_n\} \neq s$.

   define the *abstract operation* as

$$\mathsf{abstract}(P, \gamma, \delta) = P\{\gamma{:=}\langle \textbf{let } x_1{=}s_1, \ldots, x_n{=}s_n \textbf{ in } s, \mathcal{R}\rangle \multimap\}$$

2. For $\delta \in \mathsf{dom}(P)$ with $P(\delta) = \langle f(t_1, \ldots, t_n), \mathcal{R}\rangle$ where
   (a) $f \in \mathscr{F}$, and

(b) $t_i \notin \mathscr{X}$ for some $i \leq n$.

define the *split operation* as

$$\mathsf{split}(P, \delta) = P\{\delta := \langle \textbf{let } x_1 = t_1, \ldots, x_n = t_n \textbf{ in } f(x_1, \ldots, x_n), \mathcal{R}\rangle \multimap\}$$

3. For $\delta \in \mathsf{dom}(P)$ with $P(\delta) = \langle t, \mathcal{R}\rangle$ where

   (a) $t \in \mathscr{T}$, and

   (b) $\mathcal{R} \not\equiv \bot$

   define the *drop operation* as

$$\mathsf{drop}(P, \delta) = P\{\delta := \langle t, \top\rangle \multimap\}$$

It is easy to verify that these generalisation operations indeed produce correct let-terms. Figure 6.1 depicts the effect of these operations.

6.4 Supercompilation with Generalisation

It is now possible to have let-terms in the process tree, so we will need to define how these terms can be unfolded. Therefore, the unfold rules in figure 5.1 are extended with the following rule:

$$(13)\ \frac{}{\langle \textbf{let } x_1 = t_1, \ldots, x_n = t_n \textbf{ in } t_{n+1}, \mathcal{R}\rangle \Rightarrow_{\text{unf}} \langle t_i, \mathcal{R}\rangle}\ i \in \{1, \ldots, n+1\}$$

The abstract and split operations are defined for terms in $\mathscr{T}$ only, so we will have to treat nodes containing let-terms as special nodes. The let-terms are simply a way to keep the process tree from falling apart when generalisation is carried out – in this respect they are very similar to constructor nodes, *i.e.* terms with an outermost constructor. We will see that these two kinds of nodes always can be unfolded without causing non-termination, and we will therefore call such nodes *trivial*.

Definition 78 (Trivial)
A pair $\langle t, \mathcal{R}\rangle$ is *trivial* if one of the following conditions hold

1. $t = c(t_1, \ldots, t_n)$, where $c \in \mathscr{C}$ and $n \geq 1$.
2. $t \in \mathscr{L} \setminus \mathscr{T}$.

Definition 79 (Non-trivial ancestors)
Let $P \in \mathsf{T}(\mathscr{T} \times \mathscr{R})$ and $\delta \in \mathsf{dom}(P)$. Define

$$\mathsf{nontrivanc}(P, \delta) = \begin{cases} \emptyset & \text{if } P(\delta) \text{ is trivial} \\ \{\gamma \mid \gamma \in \mathsf{propanc}(\delta) \& \neg \mathsf{trivial}(\gamma)\} & \text{otherwise} \end{cases}$$

We will now abandon the simple strategy that the leaves in the process tree are be the nodes of concern. We will instead introduce the notion of a *frontier* in the process tree, which will ensure that driving is not stopped too early. First we will need some additional notation for trees, which will anticipate that we will need to consider the more general set $\mathsf{T}_\infty(\mathscr{L})$ of infinite trees (which includes the set of finite trees $\mathsf{T}(\mathscr{L})$).

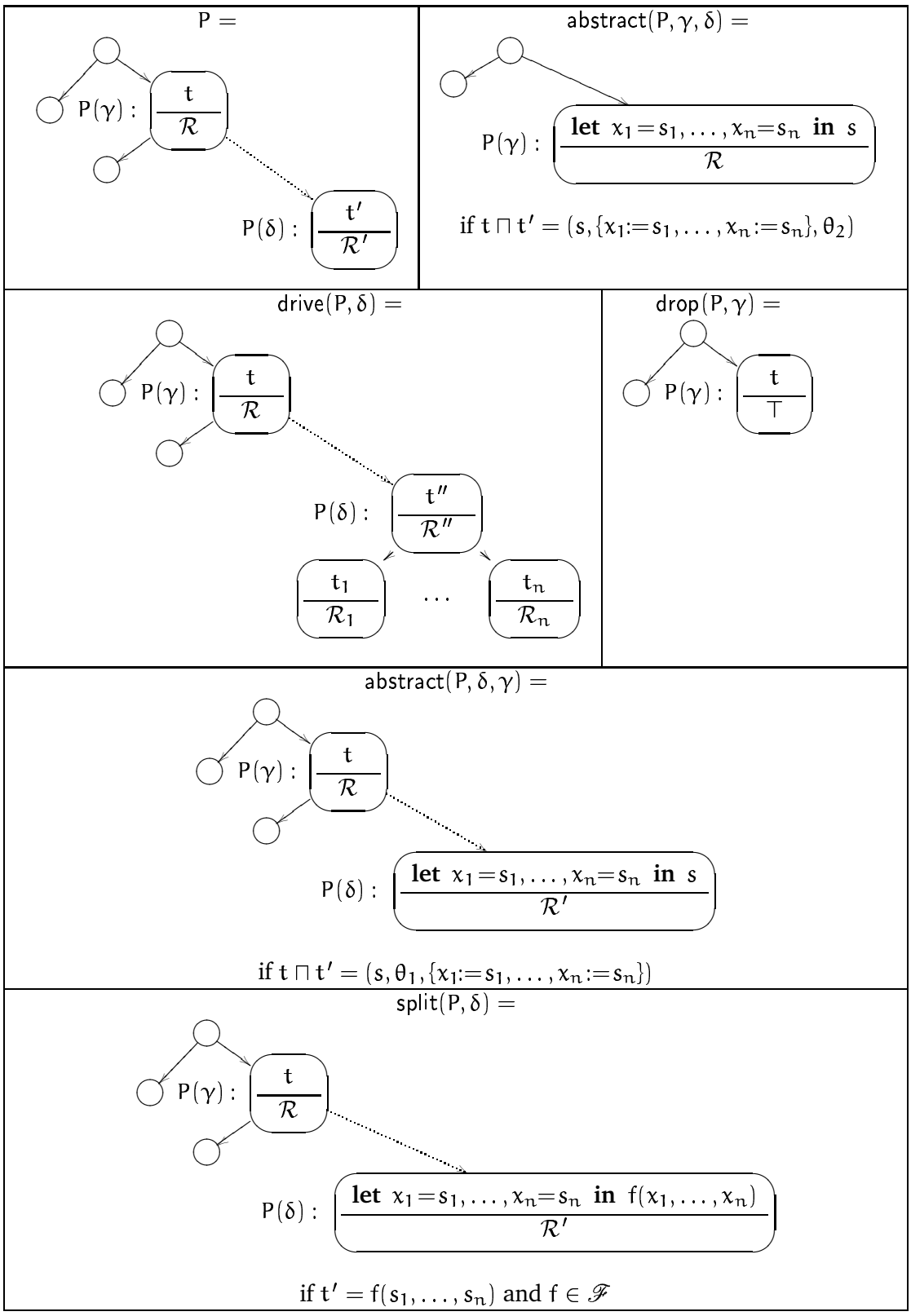
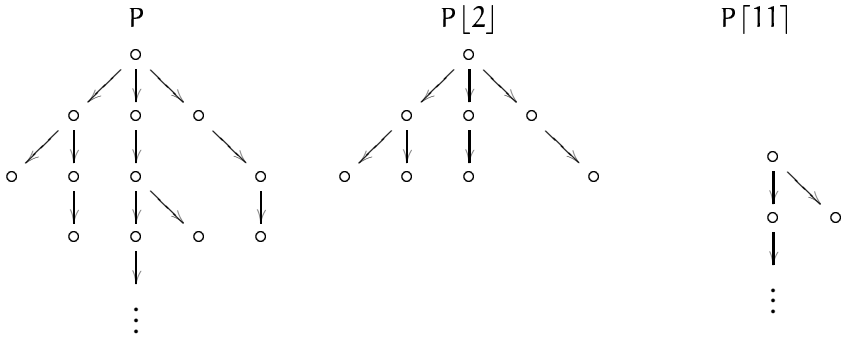


Figure 6.1: Operations used in Perfect Supercompilation

Definition 80
Let $i \in \mathbb{N}$, let $\delta, \gamma \in \mathbb{N}^*$, let $\mathcal{E}$ be a set of symbols, and let $P, Q \in T_\infty(\mathcal{E})$.

1. The *depth* $|\delta|$ of a node in P is: $\begin{array}{rcl} |\epsilon| & = & 0 \\ |\delta i| & = & |\delta| + 1 \end{array}$
2. The *depth* $|P|$ of P is: $|P| = \begin{cases} \max\{|\delta| \mid \delta \in \mathsf{dom}(P)\} & \text{if } P \text{ is finite} \\ \infty & \text{otherwise} \end{cases}$
3. The *initial subtree of depth* l *of* P, written $P\lfloor l \rfloor$, is the tree Q with
$$\begin{array}{rcl} \mathsf{dom}(Q) & = & \{\delta \in \mathsf{dom}(P) \mid |\delta| \leq l\} \\ Q(\delta) & = & P(\delta) \text{ for all } \delta \in \mathsf{dom}(Q) \end{array}$$
4. The *subtree of* P *at* δ, written $P\lceil \delta \rceil$, is the tree Q with
$$\begin{array}{rcl} \mathsf{dom}(Q) & = & \{\gamma \mid \delta\gamma \in \mathsf{dom}(P)\} \\ Q(\gamma) & = & P(\delta\gamma) \text{ for all } \gamma \in \mathsf{dom}(Q) \end{array}$$

Example 81
This example illustrates the additional operations on trees:



Definition 82
A *frontier tree* $P \in T^F_\infty(\mathcal{E})$ over some set of symbols $\mathcal{E}$ is a possibly infinite tree where a subset $\mathsf{frontier}(P) \subseteq \mathsf{dom}(P)$ of the nodes are marked and for all $\delta \in \mathsf{frontier}(P)$

1. $\mathsf{propanc}(P, \delta) \cap \mathsf{frontier}(P) = \emptyset$, and
2. $P\lceil \delta \rceil$ is finite.

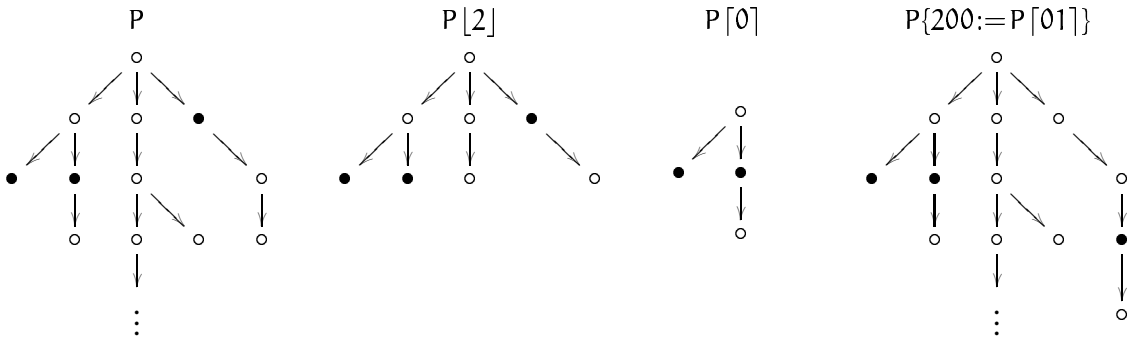
All the usual operations on trees from definition 47 carry over to frontier trees. For clarity, we repeat the definitions that affect the frontier.

1. The initial subtree of depth l of P, written $P\lfloor l \rfloor$, is the tree Q with
$$\begin{array}{rcl} \mathsf{frontier}(Q) & = & \{\delta \in \mathsf{frontier}(P) \mid |\delta| \leq l\} \\ \mathsf{dom}(Q) & = & \{\delta \in \mathsf{dom}(P) \mid |\delta| \leq l\} \\ Q(\delta) & = & P(\delta) \text{ for all } \delta \in \mathsf{dom}(Q) \end{array}$$
2. The subtree of P at δ, written $P\lceil \delta \rceil$, is the tree Q with
$$\begin{array}{rcl} \mathsf{frontier}(Q) & = & \{\gamma \mid \delta\gamma \in \mathsf{frontier}(P)\} \\ \mathsf{dom}(Q) & = & \{\gamma \mid \delta\gamma \in \mathsf{dom}(P)\} \\ Q(\gamma) & = & P(\delta\gamma) \text{ for all } \gamma \in \mathsf{dom}(Q) \end{array}$$

3. For $\delta \in \mathsf{dom}(P), P\{\delta{:=}Q\}$ denotes the tree P':

$$
\begin{array}{lcl}
\mathsf{frontier}(P') & = & \begin{cases} (\mathsf{frontier}(P) \setminus \{\gamma\}) \cup \{\delta\mu \mid \mu \in \mathsf{frontier}(Q)\} & \text{if } \gamma \in \mathsf{anc}(P,\delta) \,\&\, \gamma \in \mathsf{frontier}(P) \\ (\mathsf{frontier}(P) \setminus \{\delta\gamma \mid \delta\gamma \in \mathsf{frontier}(P)\}) \cup & \text{otherwise} \\ \qquad \{\delta\mu \mid \mu \in \mathsf{frontier}(Q)\} & \end{cases} \\
\mathsf{dom}(P') & = & (\mathsf{dom}(P) \setminus \{\delta\gamma \mid \delta\gamma \in \mathsf{dom}(P)\}) \cup \{\delta\gamma \mid \gamma \in \mathsf{dom}(Q)\} \\
P''(\mu) & = & \begin{cases} P'(\gamma) & \text{if } \mu = \delta\gamma \text{ for some } \gamma \\ P(\mu) & \text{otherwise} \end{cases}
\end{array}
$$

Example 83
Intuitively, a frontier in a tree is a curve that goes through nodes in the tree. This example illustrates the additional operations on trees (coloured nodes are in the frontier):



Definition 84 (Supercompilation)
Let $t \in \mathscr{T}$ be a term. Supercompilation of t with respect to some implicit program q is accomplished as follows:

Construct a singleton tree $P = \langle t, \top\rangle \multimap$ with $\mathsf{frontier}(P) = \{\epsilon\}$ and repeatedly apply the map $M_{\text{scp}} : \mathsf{T}^F(\mathscr{L} \times \mathscr{R}) \to \mathsf{T}^F(\mathscr{L} \times \mathscr{R})$ defined in definition 85 until the frontier is empty. Then extract a new program from P.

Definition 85
Given $P \in \mathsf{T}^F(\mathscr{L} \times \mathscr{R})$, if $\mathsf{frontier}(P) = \emptyset$ then $M_{\text{scp}}(P) = P$. Otherwise, let $\delta \in \mathsf{frontier}(P)$ and proceed as follows.

$M_{scp}(P) =$
1: **if** δ is final *(close)*
  $P = P\{\delta := P(\delta)\}$
  $\mathsf{frontier}(P) = \mathsf{frontier}(P) \setminus \{\delta\}$
2: **else if** δ is trivial *(trivial)*
  $\mathsf{frontier}(P) = \mathsf{frontier}(P) \setminus \{\delta\}$
  $P = \mathsf{drivestep}(P, \delta)$
  $\mathsf{frontier}(P) = \mathsf{frontier}(P) \cup \{\delta i\}$
3: **else if** $\forall \gamma \in \mathsf{nontrivanc}(P, \delta) : \pi_1(P(\gamma)) \not\trianglelefteq \pi_1(P(\delta))$ *(drive)*
  $\mathsf{frontier}(P) = \mathsf{frontier}(P) \setminus \{\delta\}$
  if $P\lceil\delta\rceil$ is singleton
    $P = \mathsf{drivestep}(P, \delta)$
  foreach $\delta i \in \mathsf{dom}(P)$
    $\mathsf{drive}(P, \delta i)$
    $\mathsf{frontier}(P) = \mathsf{frontier}(P) \cup \{\delta i\}$
 else begin
  let $\gamma \in \mathsf{nontrivanc}(P, \delta), P(\gamma) = \langle t, \mathcal{R}\rangle, P(\delta) = \langle t', \mathcal{R}'\rangle$ and $t \trianglelefteq t'$
4: **if** $t \doteq t'$ *(drop)*
    $P = P\{\gamma := \langle t, \top\rangle \multimap\}$
    $\mathsf{frontier}(P) = \{\gamma\} \cup (\mathsf{frontier}(P) \setminus \{\mu \mid \gamma \in \mathsf{anc}(P, \mu)\})$
5: **else if** $t \lessdot t'$ *(abstract down)*
    $P = \mathsf{abstract}(P, \delta, \gamma)$
6: **else if** $t \leftrightarrow t'$ *(split)*
    $P = \mathsf{split}(P, \delta)$
7: **else** *(abstract up)*
    $P = \mathsf{abstract}(P, \gamma, \delta)$
    $\mathsf{frontier}(P) = \{\gamma\} \cup (\mathsf{frontier}(P) \setminus \{\mu \mid \gamma \in \mathsf{anc}(P, \mu)\})$
 end

The frontier serves as a work-list of nodes that need to be processed. The transformer works as follows.

1. If the unprocessed node is final (*i.e.* a variable, a 0-ary constructor or a renaming of some ancestor such that the restriction systems allow folding), its children (if any) are discarded and the node is considered processed (*i.e.* removed from the frontier).

2. Otherwise, if the node is a trivial node, a single drivestep is performed on the node, and the children are made the new frontier.

3. Otherwise, if no ancestors are embedded in the node it is ensured that it has a number of children, which are then driven and made the new frontier.

4. Otherwise, if there exists an ancestor which is a renaming for which folding could not be carried out, the ancestor is made the new frontier and its restriction system is made empty.

5. Otherwise, if the node is a proper instance of some ancestor, it is generalised (downwards abstraction) and made the new frontier.

6. Otherwise, if the node and the embedded ancestor have no common structure that can be shared, the node is split and made the new frontier.

7. Otherwise, the embedded ancestor is generalised (upwards abstraction) and made the new frontier.

We will now ensure that the transformer produces frontier trees.

Lemma 86
M_{scp} *is a map from* $T^F(\mathscr{L} \times \mathscr{R})$ *to* $T^F(\mathscr{L} \times \mathscr{R})$.

PROOF. By definition, the domain of M_{scp} is $T^F(\mathscr{L} \times \mathscr{R})$. We must then verify that correct let-terms are added to the tree P and that the frontier is maintained properly. We will furthermore require that there is no frontier above nodes that contain proper let-terms, *i.e.* for some frontier tree P, for all nodes $\delta \in \mathsf{dom}(P)$, if $\pi_1(P(\delta)) \in \mathscr{L}$, then $\mathsf{propanc}(P, \delta) \cap \mathsf{frontier}(P) = \emptyset$. We proceed by induction on the number of operations performed. The initial singleton tree trivially satisfies this constraint. We then proceed by case analysis on the operations performed. For brevity, let P be the tree before the operation and Q the tree after the operation.

close By the induction hypothesis P fulfils the requirements. A subtree is replaced by the root of the subtree, which was in P's frontier. Then also Q fulfil the requirements.

trivial By the induction hypothesis P fulfil the requirements. Children are added to a node in P's frontier by a single drivestep. The children will all contain terms in $\mathscr{T} \times \mathscr{R}$ by definition of the unfold rule. The frontier is then move down to these children, and thus also Q fulfil the requirements.

drive By the induction hypothesis P fulfil the requirements. Let δ be a non-trivial node in P's frontier. If $P(\delta)$ is a singleton, a single unfolding step will add new children to δ. The children will all contain terms in $\mathscr{T} \times \mathscr{R}$ by definition of the unfold rule. If δ already have children, then by the induction hypothesis these must contain nodes in $\mathscr{T} \times \mathscr{R}$, since δ is in the frontier. Thus the drive operation is well-defined. Driving is carried out by applications of the unfolding rule, which can only introduce pairs in $\mathscr{T} \times \mathscr{R}$. The frontier is then moved down to the children which fulfil the requirement. Thus Q fulfil the requirements.

drop By the induction hypothesis P fulfil the requirements. A subtree is replaced by the term from the root δ of the subtree (accompanied with an empty restriction set). Thus Q fulfil the requirements.

abstract down By the induction hypothesis P fulfil the requirements. Let δ be a node in P's frontier that such that $P(\delta) = \langle t, \mathcal{R} \rangle$ is non-trivial, and let $P(\gamma) = \langle t', \mathcal{R}' \rangle$ be a non-trivial ancestor such that $t' \lessdot t$. Let $t \sqcap t' = (s, \{x_1 := s_1, \ldots, x_n := s_n\}, \theta)$. Since γ is not final, t' is not a variable. Thus, since $t' \lessdot t$, s is not a variable. Also, since again $t' \lessdot t$, $s\{x_1 := s_1, \ldots, x_n := s_n\} \neq s$. Hence the abstract operation is well-defined and results in that the subtree at δ is replaces by a let-term and the restriction set $\mathcal{R}$. Since δ is still in the frontier, Q fulfil the requirements.

split By the induction hypothesis P fulfil the requirements. Let δ be a node in P's frontier that such that $P(\delta) = \langle t, \mathcal{R} \rangle$ is non-trivial, and let $P(\gamma) = \langle t', \mathcal{R}' \rangle$ be a non-trivial ancestor such that $t' \hookrightarrow t$. Let $t \sqcap t' = (s, \{x_1 := s_1, \ldots, x_n := s_n\}, \theta)$. Since γ is not final and non-trivial, t' must have the from $f(t_1, \ldots, t_n)$. Also, since t' is not a variable and is embedded in t, $s_i \notin \mathscr{X}$, for some i. Hence the split operation is well-defined and results in that the subtree at δ is replaces by a let-term and the restriction set $\mathcal{R}$. Since δ is still in the frontier, Q fulfil the requirements.

abstract up By the induction hypothesis P fulfil the requirements. Let δ be a node in P's frontier that such that $\mathsf{P}(\delta) = \langle t, \mathcal{R} \rangle$ is non-trivial, and let $\mathsf{P}(\gamma) = \langle t', \mathcal{R}' \rangle$ be a non-trivial ancestor such that $t' \trianglelefteq t$. Let $t' \sqcap t = (s, \{x_1 := s_1, \ldots, x_n := s_n\}, \theta)$. Since $t' \not\leftrightarrow t$, s is not a variable. Also, since $t' \not\lessdot t$, then $s\{x_1 := s_1, \ldots, x_n := s_n\} \not\equiv s$. Hence the abstract operation is well-defined and results in that the subtree at γ is replaces by a let-term and the restriction set $\mathcal{R}'$. Since γ is now placed in the frontier, Q fulfil the requirements.

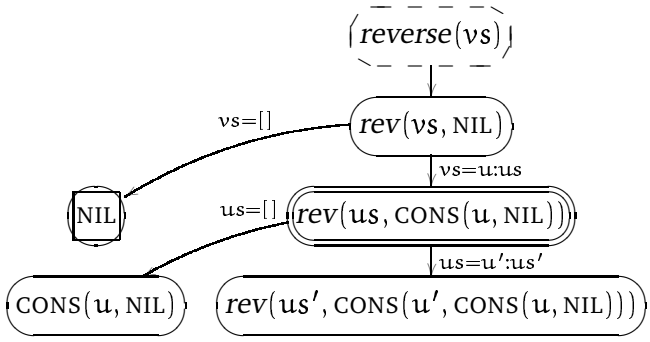
□

We will now see show that our supercompiler indeed terminates on one of the canonical examples.

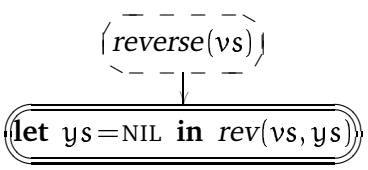
Example 87 (Accumulating Parameter Terminates)

$$
\begin{array}{lcl}
\textbf{data}\ \mathrm{List}(a) & = & \mathrm{CONS}(a, \mathrm{List}(a)) \mid \mathrm{NIL} \\
\mathit{reverse}(xs) & = & \mathit{rev}(xs, \mathrm{NIL}) \\
\mathit{rev}(\mathrm{NIL}, ys) & = & ys \\
\mathit{rev}(\mathrm{CONS}(x, xs), ys) & = & \mathit{rev}(xs, \mathrm{CONS}(x, ys))
\end{array}
$$

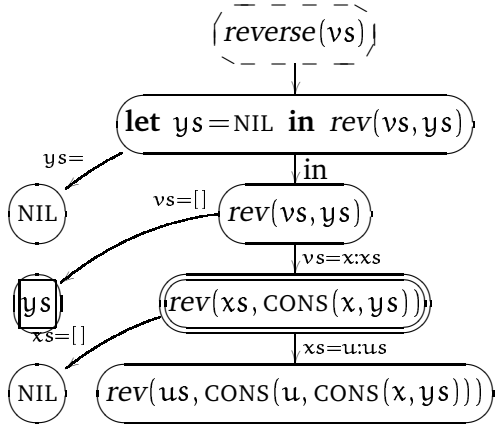
Supercompiling the term $\mathit{reverse}(vs)$ will proceed as follows. Let nodes in the frontier be marked by double nodes and locally unfolded nodes be indicated as dashed nodes. Before the first generalisation occurs, the process will look like this:



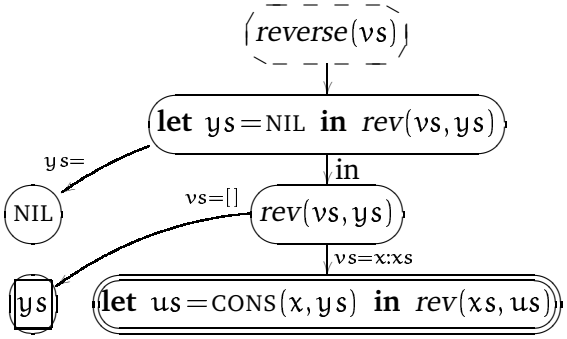
When the frontier to the right is selected, it is discovered that it embeds the parent. Since it is not an instance of the parent and they share common structure, an upwards abstraction is made.



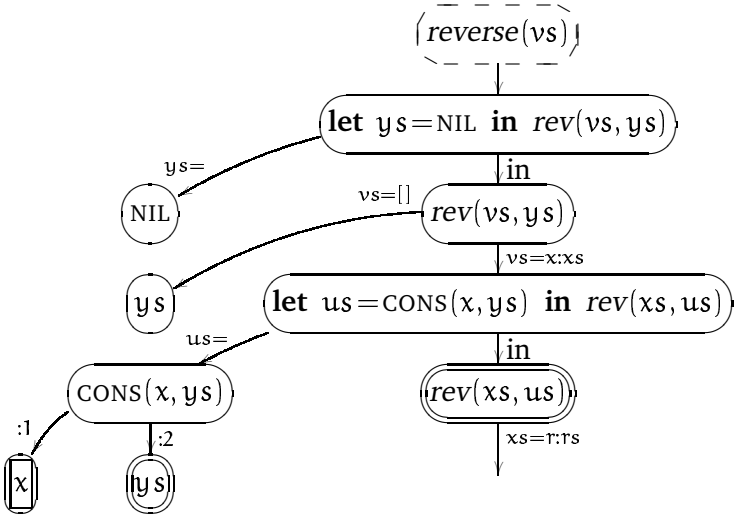
Then driving is resumed.



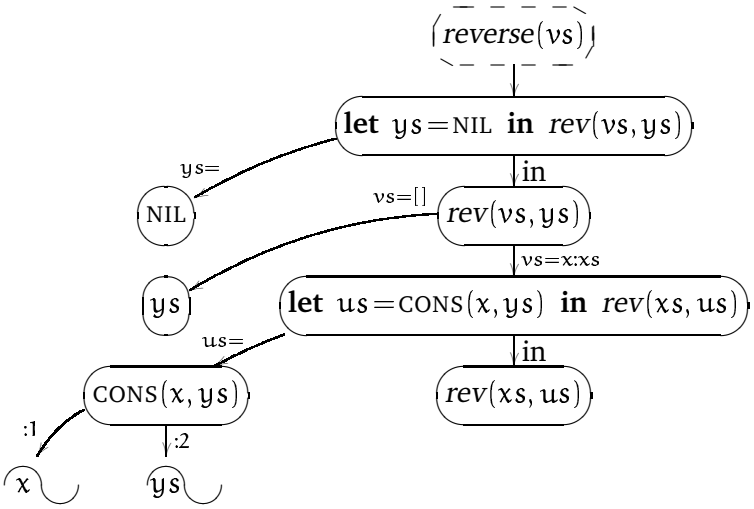
Now the node in the frontier to the right is an instance of its parent, and then a split operation is performed.



Then driving is resumed.



Finally the call to *rev* is a renaming of an ancestor and is thus final.

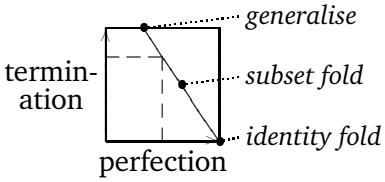


From this tree, we will get the term $f(vs)$ and the following program:

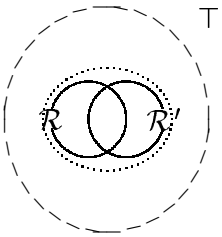
$$
\begin{array}{lcl}
\textbf{data}\ \mathrm{List}(a) & = & \mathrm{CONS}(a, \mathrm{List}(a)) \mid \mathrm{NIL} \\
f(xs) & = & g(xs, \mathrm{NIL}) \\
g(\mathrm{NIL}, ys) & = & ys \\
g(\mathrm{CONS}(x, xs), ys) & = & h(xs, \mathrm{CONS}(x, ys)) \\
h(xs, us) & = & g(xs, us)
\end{array}
$$

which is essentially the original program.

As discussed in the previous chapter, the process trees are not perfect when generalisation steps have occurred, but since we ensure termination, this is a fair trade. As termination is improved, abstractions have to be made. This relation can be depicted thus:



When we compare two restriction systems $\mathcal{R}$ and $\mathcal{R}'$, and abstractions have to made, we have chosen to simply discard the negative information, and experimental results must be conducted to show whether this is a problem in practice. If our strategy poses a problem, we will have to a develop a method that calculates a non-trivial generalisation of two restriction system:



The goal should be to get "as little as possible of the surrounding solutions" into the abstraction as illustrated above by the dotted set. If a bigger set is chosen, a greater number of solutions which never arise will be included, which will degenerate the process tree.

6.5 Summary

We have presented a supercompiler algorithm that propagates both positive and negative information. The core of the algorithm aggressively unfolds local process graphs and inserts the "interesting" nodes into a global process graph, thereby eliminating intermediate computations. When the local unfolding endangers termination, an outer algorithm performs the necessary generalisation by splitting terms into subcomponents such that driving can be resumed. In the next chapter we will prove that our supercompiler algorithm indeed terminates.

Chapter 7

Termination

Program transformation techniques such as supercompilation, deforestation or partial evaluation will always be faced with the problem of non-termination, since at the very core of any such techniques, we will find an unfold or execution step.

In the previous chapter we presented an algorithm for supercompilation that ensured termination on selected examples by means of generalisation. We will now show that this algorithm indeed terminates for any input. We consider this property to be one of the merits in this thesis, since, of the many other aggressive program transformers that have been presented in the last two decades, very few of them are guaranteed to terminate. In particular, no transformer that utilises negative information has been guaranteed to terminate.

This chapter is heavily based on work by Sørensen [25], and the bulk of definitions and propositions presented are adopted from this paper.

7.1 Abstract Program Transformers

We will now take a step back and develop a more general framework for proving termination for a large class of program transformers. As suggested in previous chapters, a program transformer can be viewed as a map from trees to trees, expressing a single step of transformation.

Definition 88 (Abstract program transformer)
An *abstract program transformer* on some set $\mathcal{E}$, is a map $M : \mathsf{T}(\mathcal{E}) \to \mathsf{T}(\mathcal{E})$.

Program transformation can then be achieved by iterative application of an abstract program transformer M to some tree P. Iterative application will be denoted as follows:

Definition 89
For $M : \mathsf{T}(\mathcal{E}) \to \mathsf{T}(\mathcal{E})$ and $P \in \mathsf{T}(\mathcal{E})$, define

$$\begin{aligned} M^0(P) &= P \\ M^{i+1}(P) &= M^i(M(P)) \end{aligned}$$

The transformation will have reached the final result when M returns its input unchanged, *i.e.* when $M(P) = P$.

Definition 90 (Termination of transformers)

1. An abstract program transformer M on $\mathsf{T}(\mathcal{E})$ *terminates on* $P \in \mathsf{T}(\mathcal{E})$ if $M^i(P) = M^{i+1}(P)$ for some $i \in \mathbb{N}$.

2. An abstract program transformer M *terminates* if M terminates on all singleton $P \in \mathsf{T}(\mathcal{E})$.

7.2 The Metric Space of Trees

From definition 90 and the previous chapters, it seems like termination of an abstract program transformer will require that the iterative process of transformation converges to the final result. If we view the transformation process as a series of steps that each produce a new tree, the above notion will amount to ensure some form of convergence of the sequence of trees produced.

A general framework for studying convergence is the theory of *metric spaces*. We will now review some properties from this theory and then show how the set of trees can be viewed as a metric space.

Definition 91 (Metric space)
Let X be a set and $d : X \times X \to \mathbb{R}_+$ be a map such that, for all $x, y, z \in X$:

1. $d(x, y) = 0 \Leftrightarrow x = y$
2. $d(x, y) = d(y, x)$
3. $d(x, y) + d(y, z) \geq d(x, z)$

Then (X, d) is a metric space.

Example 92
Let $X = \{(a, b) \mid a, b \in \mathbb{R}\}$ be the set of points in the plane and let the distance $d(p, q)$ between two points p and q be defined as the length of the vector from p to q. Then (X, d) is a metric space.

Definition 93 (Stabilise, Convergence, Cauchy)
Let (X, d) be a metric space. Then

1. A sequence $x_0, x_1, \ldots \in X$ *stabilises to* $x \in X$ if there exists an $N \in \mathbb{N}$ such that, for all $n \geq N$, $d(x_n, x) = 0$.
2. A sequence $x_0, x_1, \ldots \in X$ is *convergent with limit* $x \in X$ if, for all $\epsilon > 0$, there exists an $N \in \mathbb{N}$ such that, for all $n \geq N$, $d(x_n, x) \leq \epsilon$.
3. A sequence $x_0, x_1, \ldots \in X$ is a *Cauchy sequence* if, for all $\epsilon > 0$, there exists an $N \in \mathbb{N}$ such that, for all $n, m \geq N$, $d(x_n, x_m) \leq \epsilon$.

Any sequence has at most one limit. Also, stabilization implies convergence, and convergence implies Cauchy.

Definition 94 (Complete metric space)
Let (X, d) be a metric space. If every Cauchy sequence in (X, d) is convergent, then (X, d) is *complete*.

Definition 95 (Continuous map)
Let (X, d_X) and $(X', d_{X'})$ be metric spaces. A map $p : X \to X'$ is *continuous at* $x \in X$ if, for every sequence $x_0, x_1, \ldots \in X$ that converges to x, it holds that $p(x_0), p(x_1), \ldots \in X'$ converges to $p(x)$. Also, p is *continuous* if p is continuous at every $x \in X$.

Lemma 96 (Composition)
Let (X, d_X), $(X', d_{X'})$ and $(X'', d_{X''})$ be metric spaces. If $f : X \to X'$ and $g : X' \to X''$ are both continuous, then so is $g \circ f : X \to X''$.

We will now return to the set of trees and show that it can indeed be viewed as a metric space by the following definition of the distance between two trees.

Definition 97 (Distance for trees)
The distance $d : T_\infty(\mathcal{E}) \times T_\infty(\mathcal{E}) \to \mathbb{R}_+$ is defined as

$$d(P, P') = \begin{cases} 0 & \text{if } P = P' \\ 2^{-\min\{l \mid P[l] \neq P'[l]\}} & \text{otherwise} \end{cases}$$

Proposition 98
$(T_\infty(\mathcal{E}), d)$ *is a metric space.*

Proposition 99 (Bloom,Elgot,Wright)
The metric space $(T_\infty(\mathcal{E}), d)$ *is complete.*

7.3 Termination of Abstract Transformers

So far, we have just established a lot of facts about the metric space of trees. We will now use these properties to establish a set of condition that will ensure that an abstract program transformer terminates.

The idea is to ensure that the transformer incrementally produces trees that are still more stable, in the sense that consecutive trees are equal to some increasing depth; and at the same time ensure that this process does not go on indefinitely, *i.e.* that "no infinite trees will be produced".

Definition 100 (Finitary)
A predicate $p : T_\infty(\mathcal{E}) \to \mathbb{B}$ is *finitary* if $p(P) = \text{FALSE}$ for every infinite tree P.

Lemma 101 (Continuous predicate)
A predicate p *on* $T_\infty(\mathcal{E})$ *is continuous if and only if, for every convergent sequence* $P_0, P_1, \ldots \in T_\infty(\mathcal{E})$ *with limit* P*, the sequence* $p(P_0), p(P_1), \ldots$ *stabilises to* $p(P)$*.*

Definition 102 (Cauchy transformer)
An abstract program transformer M on $\mathcal{E}$ is *Cauchy* if, for every singleton $P \in T(\mathcal{E})$, the sequence $P, M(P), M^2(P), \ldots$ is a Cauchy sequence.

Definition 103 (Maintaining a predicate)
Let $M : T(\mathcal{E}) \to T(\mathcal{E})$ be an abstract program transformer on a set of symbols $\mathcal{E}$ and let $p : T_\infty(\mathcal{E}) \to \mathbb{B}$ be a predicate. M *maintains* p if, for every $P \in T(\mathcal{E})$ it holds that $p(M^i(P)) = \text{TRUE}$ for infinitely many $i \in \mathbb{N}$.

Theorem 104 (Sørensen's Theorem)
Let an abstract program transformer $M : T(\mathcal{E}) \to T(\mathcal{E})$ *maintain predicate* $p : T_\infty(\mathcal{E}) \to \mathbb{B}$*. If*

1. M *is Cauchy, and*
2. p *is finitary and continuous,*

then M *terminates.*

PROOF. See [25]. □

We will now show that our supercompilation transformer M_{scp} (definition 85) terminates by using the above theorem. This requires that the sequence of trees produced is a Cauchy sequence, and that a predicate is maintained such that "no infinite tree is produced". For this end, we impose a set of orderings on trees. Let us therefore review some properties about quasi-orderings.

Definition 105 (Quasi-orders)
Let $\mathcal{E}$ be a set with a relation $\preceq$. Then $(\mathcal{E}, \preceq)$ is a *quasi-order* if $\preceq$ is transitive and reflexive. We write $e \prec e'$ if $e \preceq e'$ and $e' \not\preceq e$, for any $e, e' \in \mathcal{E}$.

Let $(\mathcal{E}, \preceq)$ be a quasi-order.

1. $(\mathcal{E}, \preceq)$ is *well-founded* if there is no infinite sequence $e_1, e_2, \ldots \in \mathcal{E}$ with $e_1 \succ e_2 \succ \ldots$.
2. $(\mathcal{E}, \preceq)$ is a *well-quasi-order* if, for every infinite sequence $e_1, e_2, \ldots \in \mathcal{E}$, there are $i < j$ with $e_i \preceq e_j$.

Theorem 106 (Kruskal's Tree Theorem)
The relation $\trianglelefteq$ on $T(\mathscr{T})$ is a well-quasi-order.

PROOF. See [25]. □

The theorem above will be used to prove that no infinite trees will be produced by our supercompiler, but it cannot be used directly since driving is resumed after each generalisation step. We will therefore need more sophisticated orderings to prove termination.

In the rest of this section we will show that our supercompiler indeed fits the criteria for termination. Let us start with proving that driving always stops.

Lemma 107 (Drive terminates)
Algorithm drive : $T(\mathscr{T} \times \mathscr{R}) \times \mathbb{N}^* \to T(\mathscr{T} \times \mathscr{R})$ terminates. (algorithm 72)

PROOF. The algorithm incrementally constructs a tree Q from an initial singleton. The development of Q will certainly terminate if, at any time, more that one leaf is appended. Let us then assume that it generates an infinite branch. By Kruskal's Tree Theorem (theorem 106), $\trianglelefteq$ is a well-quasi-order and thus in any infinite branch there will be $\gamma < \delta$ such that $P(\gamma) \trianglelefteq P(\delta)$, which would force the algorithm to stop at δ. Hence it terminates. □

Let us start with a more specific abstract transformer, namely one that maintains a frontier.

Proposition 108 (Frontier transformers are Cauchy)
Let $(\mathcal{E}, \preceq)$ be a well-founded quasi-order, and $M : T^F(\mathcal{E}) \to T^F(\mathcal{E})$ be such that, for all $P \in T^F(\mathcal{E})$, $M(P) = P\{\delta := Q\}$ for some $\delta \in dom(P)$ and $Q \in T^F(\mathcal{E})$, where either

1. *$\delta \in frontier(P)$, $P(\delta) = Q(\epsilon)$, and $\delta \notin frontier(P\{\delta := Q\})$ (unfold); or*
2. *$\exists \gamma : \delta\gamma \in frontier(P)$, and $P(\delta) \succ Q(\epsilon)$ and $\delta \in frontier(P\{\delta := Q\})$ (generalise).*

Then M is Cauchy.

PROOF. Given a tree $P \in T(\mathcal{E})$, let $P_i = M^i(P)$. We must prove that, for any depth $l \in \mathbb{N}$, there exists $N \in \mathbb{N}$ such that, for all $n, m \geq N$, $P_n\lfloor l \rfloor = P_m\lfloor l \rfloor$. We do this by induction on the depth l:

For $l =$ FALSE, then either

1. $\mathsf{frontier}(P) = \emptyset$, *i.e.* there is no frontier in the P. Then neither unfolding nor generalisation can take place, and thus $P_n\lfloor 0 \rfloor = P_m\lfloor 0 \rfloor$ for all $n, m \in \mathbb{N}$, or

2. $\mathsf{frontier}(P) \neq \emptyset$. Suppose now that $\{P_i(\epsilon) \mid i \in \mathbb{N}\}$ is infinite, *i.e.* there are infinitely many different roots. Since no unfolding can change the root, there must be infinitely many generalization steps at the root, *i.e.* for infinitely many i, $P_i(\epsilon) \succ P_{i+1}(\epsilon)$. This contradicts the assumption that $\preceq$ is a well-founded quasi-order. Hence there is an N_0 such that, for all $n, m \geq N_0$, $P_n\lfloor 0 \rfloor = P_m\lfloor 0 \rfloor$.

For $l > 0$, by the induction hypothesis there is an N_{l-1} such that, for all $n, m \geq N_{l-1}$, $P_n\lfloor l-1 \rfloor = P_m\lfloor l-1 \rfloor$. Thus no generalization step can be performed at levels $0, \ldots, l-1$, since such a generalization would change a node. Assume now that a series of unfold and generalisation steps take place. Each unfold step performed on a node in the frontier at levels $0, \ldots, l-1$ will move the frontier downwards or remove some part of it. There must then be an $M \geq N_0$ such that, for all $m \geq M$, no more unfolding steps are carried out at levels $0, \ldots, l-1$. Thus there will be a number K such that, for all $m \geq M$, the number of children at level l will be K. For each such child γ_i where $i \in \{1, \ldots, K\}$, we can then proceed with $P\lceil \gamma_i \rceil$ as in the case of $l = 0$, since no unfold or generalisation steps can be performed at level $0, \ldots, l-1$. This will give us K numbers M_i such that, for all $n, m \geq M_i$, $P_n\lceil \gamma_i \rceil\lfloor 0 \rfloor = P_m\lceil \gamma_i \rceil\lfloor 0 \rfloor$. Let $N = \max\{M_i\}$. Then, for all $n, m \geq N$, $P_n\lfloor l \rfloor = P_m\lfloor l \rfloor$. □

Definition 109
Let $|\mathcal{R}|$ denote the size of $\mathcal{R}$. Define a relation $\succeq$ on $(\mathscr{L} \times \mathscr{R})$ by

$$\langle v, \mathcal{R} \rangle \succeq \langle v', \mathcal{R}' \rangle \Leftrightarrow \begin{pmatrix} v \in \mathscr{T} \ \vee \ v' \in \mathscr{L} \setminus \mathscr{T} \\ \text{or} \\ |\mathcal{R}| \geq |\mathcal{R}'| \end{pmatrix}$$

We have that $\langle v, \mathcal{R} \rangle \succ \langle v', \mathcal{R}' \rangle \Leftrightarrow (v \in \mathscr{T} \wedge v' \in \mathscr{L} \setminus \mathscr{T}) \vee (|\mathcal{R}| > |\mathcal{R}'|)$. This means in particular that replacing a regular term with a let-term or replacing a non-empty restriction set with an empty restriction set strictly decreases the order.

Lemma 110
$(\mathscr{L}, \succeq)$ is a well-founded quasi-order.

PROOF. Easy, since $>$ is a well-founded quasi-order. □

Proposition 111
M_{scp} is Cauchy.

PROOF. We now show that, for any $P \in \mathsf{T}^F(\mathscr{L})$, $M_{\mathrm{scp}}(P) = P\{\delta := Q\}$ for some $\delta \in \mathsf{dom}(P)$ and $Q \in \mathsf{T}^F(\mathscr{L})$, either

1. $\delta \in \mathsf{frontier}(P)$, $P(\delta) = Q(\epsilon)$, and $\delta \notin \mathsf{frontier}(P\{\delta := Q\})$; or

2. $\exists \gamma : \delta\gamma \in \mathsf{frontier}(P)$, $P(\delta) \succ Q(\epsilon)$ and $\delta \in \mathsf{frontier}(P\{\delta := Q\})$.

We do this by case analysis on the operations performed by M_{scp}.

close $\delta \in \mathsf{frontier}(P)$ and $M_{\mathrm{scp}}(P) = P\{\delta := P(\delta) \multimap\}$. Trivially, $P(\delta) = Q(\epsilon)$, and since δ is removed from the frontier, $\delta \notin \mathsf{frontier}(P\{\delta := Q\})$.

trivial $\delta \in \mathsf{frontier}(P)$. A single unfolding step will add new children to δ, which will not change $P(\delta)$. Thus $P(\delta) = P\{\delta := Q\}(\delta)$. Since δ is removed from the frontier and each δi put in the frontier, $\delta \notin \mathsf{frontier}(P\{\delta := Q\})$.

drive $\delta \in \mathsf{frontier}(P)$. If $P(\delta)$ is a singleton, a single unfolding step will add new children to δ, which will not change $P(\delta)$ and not affect the frontier. In any case, each child δi will be driven by algorithm drive (which terminates by lemma 107). Thus $P(\delta) = P\{\delta := Q\}(\delta)$. Since δ is removed from the frontier and each δi put in the frontier, $\delta \notin \mathsf{frontier}(P\{\delta := Q\})$.

drop Since $\delta \in \mathsf{propanc}(\mu)$ for some $\mu \in \mathsf{frontier}(P)$, $\exists \gamma : \delta\gamma \in \mathsf{frontier}(P)$. Let $P(\mu) = \langle t, \mathcal{R} \rangle$, $P(\delta) = \langle t', \mathcal{R}' \rangle$ and $t \doteq t'$. Since μ is not final, it has not been possible to fold back to δ. Thus $\mathcal{R}'$ must be different from $\top$. Thus replacing $\langle t', \mathcal{R}' \rangle$ with $\langle t', \top \rangle$ strictly decreases the size of the restriction system. Hence $P(\delta) \succ Q(\epsilon)$. Finally, δ is put in the frontier which means that $\delta \in \mathsf{frontier}(P\{\delta := Q\})$.

abstract down Since $\delta \in \mathsf{frontier}(P)$, trivially $\exists \gamma : \delta\gamma \in \mathsf{frontier}(P)$. By definition of the abstract operation, $P(\delta) \in (\mathscr{T} \times \mathscr{R})$. Thus replacing $P(\delta)$ with a let-term strictly decreases the order, so $P(\delta) \succ Q(\epsilon)$. Finally, δ is put in the frontier which means that $\delta \in \mathsf{frontier}(P\{\delta := Q\})$.

split As downward abstraction.

abstract up Since $\delta \in \mathsf{propanc}(\mu)$ for some $\mu \in \mathsf{frontier}(P)$, $\exists \gamma : \delta\gamma \in \mathsf{frontier}(P)$. By definition of the abstract operation, $P(\delta) \in (\mathscr{T} \times \mathscr{R})$. Thus replacing $P(\delta)$ with a let-term strictly decreases the order, so $P(\delta) \succ Q(\epsilon)$. Finally, δ is put in the frontier which means that $\delta \in \mathsf{frontier}(P\{\delta := Q\})$.

Then use proposition 108. □

We will now show that our supercompiler M_{scp} maintains a finitary and continuous predicate. We will again begin with some abstract definitions and then use properties about these abstract definitions.

Definition 112 (Finite character)
A predicate $p : T_\infty(\mathcal{E}) \to \mathbb{B}$ is of *finite character* if and only if, for all $P \in T_\infty(\mathcal{E})$:

$$p(P) = \text{TRUE} \Leftrightarrow \forall l \in \mathbb{N} : p(P\lfloor l \rfloor) = \text{TRUE}$$

Observe that this definition implies that

$$p(P) = \text{FALSE} \Leftrightarrow \exists l \in \mathbb{N} : p(P\lfloor l \rfloor) = \text{FALSE}$$

Proposition 113
Suppose $p : T_\infty(\mathcal{E}) \to \mathbb{B}$ *is finitary and of finite character. Then* p *is continuous.*

PROOF. See [25]. □

Proposition 114
Let $\{E_1, E_2\}$ *be a partition of the set* $\mathcal{E}$*,* $\preceq_1$ *be a well-quasi-order on* E_1*,* $\preceq_2$ *be a well-founded quasi-order on* E_2*, and let* $p : T_\infty(\mathcal{E}) \to \mathbb{B}$ *be defined by*

$$p(P) = \begin{cases} \text{FALSE} & \textit{if } \exists \delta, \gamma \in \textit{dom}(P) : \delta \in \textit{propanc}(P, \gamma) \,\&\, P(\delta), P(\gamma) \in E_1 \,\&\, P(\delta) \preceq_1 P(\gamma) \\ \text{FALSE} & \textit{if } \exists \delta, \delta i \in \textit{dom}(P) : P(\delta), P(\delta i) \in E_2 \,\&\, P(\delta) \not\preceq_2 P(\delta i) \\ \text{TRUE} & \textit{otherwise} \end{cases}$$

Then p *is finitary and continuous.*

PROOF. Since trees are finitely branching, an infinite tree P must have an infinite branch by König's Lemma. Such an infinite branch will contain either

1. an infinite sequence of symbols from E_2, and thus, since $\preceq_2$ is a well-founded quasi-order, there must then be some $\delta, \delta i$ such that $\delta \not\preceq_2 \delta i$; or
2. a sequence with infinitely many symbols from E_1, and thus, since $\preceq_1$ is a well-quasi-order, there must then be some δ, γ where $\delta \in \mathsf{propanc}(P, \gamma)$ such that $\delta \preceq_1 \gamma$.

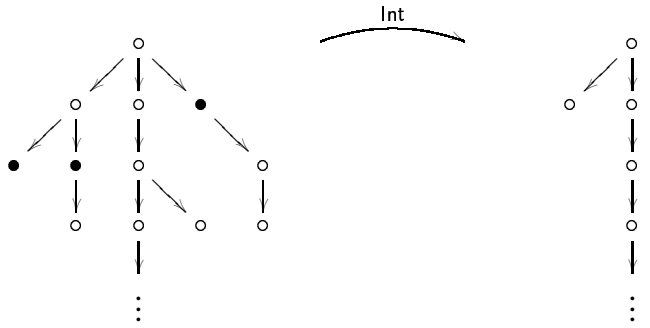
In any case, $p(P) = \text{FALSE}$, so p is finitary.

To prove p is continuous, we use definition 112. If $p(P) = \text{TRUE}$ then cleanly also $p(P\lfloor l \rfloor) = \text{TRUE}$ for all l. Conversely, if for all l it holds that $p(P\lfloor l \rfloor) = \text{TRUE}$, then also $p(P) = \text{TRUE}$. Indeed, if $p(P) = \text{FALSE}$, then already $p(P\lfloor l \rfloor) = \text{FALSE}$, for some l. □

Definition 115 (Interior)
The function $\mathsf{Int} : \mathsf{T}^F_\infty(\mathcal{E}) \to \mathsf{T}_\infty(\mathcal{E})$ gives the *interior* of some $P \in \mathsf{T}^F_\infty(\mathcal{E})$, and is defined by

1. $\mathsf{dom}(\mathsf{Int}(P)) = (\mathsf{dom}(P) \setminus (\mathsf{leaf}(P) \cup \{\delta \mid \mathsf{anc}(P, \delta) \cap \mathsf{frontier}(P) \neq \emptyset\})) \cup \{\epsilon\}$.
2. $\mathsf{Int}(P)(\delta) = P(\delta)$ for all $\delta \in \mathsf{dom}(\mathsf{Int}(P))$.

Example 116
Let the frontier in a tree be indicated by coloured nodes. The Int function will map the tree on the left-hand side to the tree on the right-hand side:



Proposition 117
Let $p : \mathsf{T}_\infty(\mathcal{E}) \to \mathbb{B}$ be finitary and continuous. Then also the map $q : \mathsf{T}^F_\infty(\mathcal{E}) \to \mathbb{B}$ defined by

$$q(P) = p(\mathit{Int}(P))$$

is finitary and continuous.

PROOF. For infinite P, $\mathsf{Int}(P)$ is also infinite since every subtree removed by Int is finite. Thus $q(P) = p(\mathsf{Int}(P)) = \text{FALSE}$, and hence q is finitary.

To show q is continuous, it is sufficient (by lemma 96) to show that $\mathsf{Int} : \mathsf{T}^F_\infty(\mathcal{E}) \to \mathsf{T}_\infty(\mathcal{E})$ is continuous. Now, if $P_1, P_2, \ldots$ converges to P, there is, for any $l \in \mathbb{N}$, an $N \in \mathbb{N}$ such that for all $n \geq N$, $P_n\lfloor l+1 \rfloor = P\lfloor l+1 \rfloor$. Then $\mathsf{frontier}(P_n\lfloor l+1 \rfloor) = \mathsf{frontier}(P\lfloor l+1 \rfloor)$, but $\mathsf{leaf}(P_n\lfloor l+1 \rfloor)$ might be different from $\mathsf{leaf}(P\lfloor l+1 \rfloor)$. Thus $\mathsf{Int}(P_n)\lfloor l \rfloor = \mathsf{Int}(P)\lfloor l \rfloor$. Hence $\mathsf{Int}(P_1)\lfloor l \rfloor, \mathsf{Int}(P_2)\lfloor l \rfloor, \ldots$ converges to $\mathsf{Int}(P)$ as required. □

We will now define a well-founded quasi-order on trivial terms to be able to use the above propositions.

Definition 118 (Size of terms)

1. Define $\mathsf{norm} : \mathscr{L} \to \mathscr{T}$ by

$$\begin{array}{lcl} \mathsf{norm}(\textbf{let } x_1{=}t_1, \ldots, x_n{=}t_n \textbf{ in } t) & = & t\{x_1{:=}t_1, \ldots, x_n{:=}t_n\} \\ \mathsf{norm}(t) & = & t \end{array}$$

2. Define the *size* $|\bullet| : \mathscr{L} \to \mathbb{N}$ by

$$\begin{array}{lcl} |\textbf{let } x_1{=}t_1, \ldots, x_n{=}t_n \textbf{ in } t| & = & |\mathsf{norm}(\textbf{let } x_1{=}t_1, \ldots, x_n{=}t_n \textbf{ in } t)| \\ |h(t_1, \ldots, t_n)| & = & 1 + |t_1| + \cdots + |t_n| \\ |\textbf{if } t_1{==}t_2 \textbf{ then } t_3 \textbf{ else } t_4| & = & 1 + |t_1| + \cdots + |t_4| \\ |x| & = & 1 \end{array}$$

3. Define $\sqsupseteq$ on $v \in \mathscr{L}$ by

$$v \sqsupseteq v' \Leftrightarrow |v| > |v'| \wedge (|v| = |v'| \,\&\, \mathsf{norm}(v) \trianglerighteq \mathsf{norm}(v'))$$

Lemma 119
The real-ation $\sqsupseteq$ is a well-founded quasi-order.

PROOF. Easy, since $\trianglerighteq$ is a well-founded quasi-order. □

Lemma 120
For all trivial $v \in \mathscr{L}$, $v \Rightarrow v'$ implies $v \sqsupset v'$.

PROOF.

1. $v = c(t_1, \ldots, t_n) \Rightarrow t_i$, for some i. Then

$$\begin{array}{rcl} |v| & = & |c(t_1, \ldots, t_n)| \\ & = & 1 + |t_1| + \cdots + |t_n| \\ & > & |t_i| \end{array}$$

   so $v \sqsupset t_i$.

2. $v = \textbf{let } x_1{=}t_1, \ldots, x_n{=}t_n \textbf{ in } t \Rightarrow v'$. We consider the two cases:

   (a) $v' = t_i$ for some i. Since $x_i \in \mathsf{Var}(t)$ and $t \neq x_i$,

$$\begin{array}{rcl} |v| & = & |t\{x_1{:=}t_1, \ldots, x_n{:=}t_n\}| \\ & > & |t_i| \\ & = & |v'| \end{array}$$

   so $v \sqsupset v'$.

   (b) $v' = t$. Clearly, $|v| \geq |t|$. If all t_i are variables or 0-ary constructors, $|v| = |v'|$, but then $t\{x_1{:=}t_1, \ldots, x_n{:=}t_n\}$ is a proper instance of t, so

$$\mathsf{norm}(v) = t\{x_1{:=}t_1, \ldots, x_n{:=}t_n\} \trianglerighteq t = \mathsf{norm}(v)$$

   so $v \sqsupset v'$.

□

Proposition 121
M_{scp} *maintains a finitary and continuous predicate.*

PROOF. Consider the predicate $q : \mathsf{T}^F_\infty(\mathscr{L} \times \mathscr{R}) \to \mathbb{B}$ defined by

$$q(P) = p(\mathsf{Int}(P))$$

where $p : \mathsf{T}_\infty(\mathscr{L} \times \mathscr{R}) \to \mathbb{B}$ is defined by

$$p(P) = \begin{cases} \text{FALSE} & \text{if } \exists \delta, \gamma \in \mathsf{dom}(P) : \ \delta \in \mathsf{propanc}(P, \gamma) \,\&\, P(\delta), P(\gamma) \text{ are non-trivial} \\ & \qquad\qquad \&\, \pi_1(P(\delta)) \trianglelefteq \pi_1(P(\gamma)) \\ \text{FALSE} & \text{if } \exists \delta, \delta i \in \mathsf{dom}(P) : P(\delta), P(\delta i) \text{ are trivial } \&\, \pi_1(P(\delta)) \not\sqsupseteq \pi_1(P(\delta i)) \\ \text{TRUE} & \text{otherwise} \end{cases}$$

The set of trivial and non-trivial nodes constitute a partition on $\mathscr{L}$ and thus also on $(\mathscr{L}, \mathscr{R})$. $\trianglelefteq$ is a well-quasi-order on non-trivial let-terms and $\sqsupseteq$ is a well-founded quasi-order on trivial let-terms. It follows from proposition 114 that p is finitary and continuous. By proposition 117 it follows that also q is finitary and continuous.

It remains to show that M_{scp} maintains q, *i.e.* that, for any singleton $P \in \mathsf{T}^F_\infty(\mathscr{L} \times \mathscr{R})$, $q(M^i_{scp}(P)) = \text{TRUE}$ for all i. We proceed by induction on i, but we now strengthen the induction hypothesis and require that all children of trivial nodes are strictly smaller than their parent, *i.e.* for all trivial nodes $\delta \in \mathsf{dom}(P_i)$ it must hold that $P_i(\delta) \sqsupset_i (\delta j)$.

For $i = 0$, P is singleton and thus there are no ancestors, so the induction hypothesis is trivially true.

For $i > 0$, we analyse each case of operation performed by M_{scp} on $M^{i-1}_{scp}(P)$. For brevity, let $P' = M^{i-1}_{scp}(P)$.

close For some node $\delta \in \mathsf{frontier}(P')$, $M_{scp}(P') = P'\{\delta := P'(\delta) \multimap\}$. Since δ is both non-trivial and in the frontier before the operation, and becomes δ a leaf afterwards, $\mathsf{Int}(M_{scp}(P')) = \mathsf{Int}(P')$. By the induction hypothesis $q(P') = \text{TRUE}$ and thus $q(M_{scp}(P')) = \text{TRUE}$. Hence M_{scp} maintains q and the induction hypothesis holds.

trivial For some trivial node $\delta \in \mathsf{frontier}(P)$, a single drivestep step will add new children to δ. Thus $M_{scp}(P')(\delta) = P'(\delta)$. Then the children are put in the frontier, and δ is removed from the frontier. By lemma 120, $M_{scp}(P')(\delta) \sqsupset M_{scp}(P')(\delta i)$ for all i. By the induction hypothesis $q(P') = \text{TRUE}$, and thus $q(M_{scp}(P')) = \text{TRUE}$. Hence M_{scp} maintains q and the induction hypothesis holds.

drive For some non-trivial node $\delta \in \mathsf{frontier}(P')$, it is first ensured that δ has children $\delta 0, \ldots, \delta n$. Then the children are driven and $\delta 0, \ldots, \delta n$ are put in the frontier, and δ is removed from the frontier. Thus $M_{scp}(P')(\delta) = P'(\delta)$ and $\mathsf{dom}(\mathsf{Int}(M_{scp}(P'))) \setminus \mathsf{dom}(\mathsf{Int}(P')) = \{\delta\}$. by the induction hypothesis $q(P') = \text{TRUE}$. Since it has already been ensured that no ancestor is embedded in $P'(\delta)$, $q(M_{scp}(P')) = \text{TRUE}$. Hence M_{scp} maintains the q, and since δ is non-trivial, the induction hypothesis holds.

drop For some non-trivial node δ above the frontier, $M_{scp}(P') = P'\{\delta := \langle t, \top \rangle \multimap\}$, where $t = \pi_1(P'(\delta))$. If the parent to δ is trivial, it was by the induction hypothesis greater than $P'(\delta)$, and it is thus also greater than $M_{scp}(P')(\delta)$. Since $\mathsf{Int}(M_{scp}(P')) = \mathsf{Int}(P') \setminus \{\delta\gamma \mid \delta\gamma \in \mathsf{dom}(P')\}$, the induction hypothesis also holds for $M_{scp}(P')$.

abstract down For some non-trivial node $\delta \in \mathsf{frontier}(P')$, $M_{scp}(P') = P'\{\delta := \langle v, \mathcal{R} \rangle\}$, where $\mathsf{norm}(v) = \pi_1(P'(\delta))$. If the parent to δ is trivial, it was by the induction hypothesis greater than $P'(\delta)$, and it is thus also greater than $M_{scp}(P')(\delta)$. Since $\mathsf{Int}(M_{scp}(P')) = \mathsf{Int}(P')$, the induction hypothesis also holds for $M_{scp}(P')$.

split As in downwards abstraction.

upwards abstraction For some non-trivial node δ above the frontier, $M_{\text{scp}}(P') = P'\{\delta := \langle \upsilon, \mathcal{R} \rangle\}$, where $\mathsf{norm}(\upsilon) = \pi_1(P'(\delta))$. If the parent to δ is trivial, it was by the induction hypothesis greater than $P'(\delta)$, and it is thus also greater than $M_{\text{scp}}(P')(\delta)$. Since $\mathsf{Int}(M_{\text{scp}}(P')) = \mathsf{Int}(P') \setminus \{\delta\gamma \mid \delta\gamma \in \mathsf{dom}(P')\}$, the induction hypothesis also holds for $M_{\text{scp}}(P')$.

□

Theorem 122
M_{scp} terminates for any singleton tree.

PROOF. By proposition 111, M_{scp} is Cauchy. By proposition 121, M_{scp} maintains a continuous and finitary predicate. Then by theorem 104, M_{scp} terminates. □

Chapter 8

Code Generation

Extracting a program from a process tree has so far been explained informally. In this chapter we will define a code-generating function $[\![\bullet]\!]$ that from a tree $P \in T(\mathscr{L})$ produces the transformed program.

In the previous chapters, the labels on edges were not represented explicitly. We will assume that the edges are labelled as described in the beginning of chapter 5. We will use the quasi-quotes $\ulcorner$ and $\urcorner$ to bracket strings of code that should appear in the transformed program. The concatenation operator $+\!\!+$ appends string of code. The code-generating function is written in continuation-passing style to ensure that code is produced in the right order: the continuation contains a "hole" that must be filled with the code produced. The root of the tree represents the initial term, so the initial continuation must wrap the call generated into a special construct that will tell us that this call, say $f(x_1, \ldots, x_n)$, is the transformed term. We will use the initial continuation $(\ulcorner \leftarrow \urcorner +\!\!+)$. Again, if the root of the process tree produced $f(x_1, \ldots, x_n)$, the the result would be a line containing $\leftarrow f(x_1, \ldots, x_n)$ in the program.

When code generation starts, there is no need for the restriction sets, which are therefore ignored in this chapter.

When generating new function definitions, we must make sure each function gets a unique name. For that end, assume that the code-generating function has access to a global, infinite set of yet unused function names F.

A leaf node that is a renaming of some ancestor in the process tree must manifest itself as a recursive call. To keep track of which function names are assigned to which nodes, a mapping E from nodes to function names is maintained by the code-generating function.

We will use the shorthand notation $\vec{y}_i$ to denote $y_1, \ldots, y_{l_i}$.

Definition 123
Let the code-generating function $[\![\bullet]\!]_E : T(\mathscr{L}) \to (\mathscr{Q} \to \mathscr{Q})$ be defined by the equations in figure 8.1 and 8.2.

We have left out the generation of user-defined data types, since these can be copied directly from the original program. We have also left out what happens if there is a stuck computation in the process tree – *i.e.* a call to a pattern function that does not have a definition for the particular pattern used in the call – which would result in a leaf that is *not* a renaming of an ancestor. In this case the code-generation function could simply generate a call to a function that always fails.

We assume that a leaf that is renamings of some ancestor – and should thus result in a call – has an annotation that can tell us the position of the ancestor. Notice that there is always exactly one such ancestor, since driving stops as soon a renaming occurs.

$$\left[\!\!\left[\ \delta :\ \boxed{\textbf{let } x_1{=}t_1,\ldots,x_n{=}t_n \textbf{ in } t'}\ \xrightarrow{\text{in}}\ t'\ \right]\!\!\right]_E =$$

$$\left[\!\!\left[\ \delta :\ \boxed{\textbf{let } x_1{=}t_1,\ldots,x_n{=}t_n \textbf{ in } t'}\ \xrightarrow{x_1=}\ t_1\ \cdots\ \xrightarrow{x_n=}\ t_n\ \right]\!\!\right]_E =$$

let $f \in F$
$F = F \setminus \{f\}$
$\{y_1,\ldots,y_m\} = \mathsf{Var}(t') \setminus \{x_1,\ldots,x_n\}$
$E = E \cup \{\delta \mapsto f\}$
in

$$\lambda\kappa.\left(\begin{array}{l}[\![\ t'\ ]\!]_E\ (\lambda t'.{\ulcorner} f({\ulcorner},\ldots,x_n,y_1,\ldots,y_m) = {\urcorner} +\!\!+ t') +\!\!+ \\ [\![\ t_1\ ]\!]_E\ (\ldots([\![\ t_n\ ]\!]_E\ (\lambda t_n\ldots t_1.\kappa\left(\begin{array}{l}{\ulcorner} f({\urcorner} +\!\!+ t_1 +\!\!+ {\ulcorner},{\urcorner} +\!\!+ \cdots +\!\!+ \\ {\ulcorner},{\urcorner} +\!\!+ t_n +\!\!+ {\ulcorner},y_1,\ldots,y_m){\urcorner}\end{array}\right))\ldots)\end{array}\right)$$

$$\left[\!\!\left[\ \delta :\ \boxed{t}\ \xrightarrow{x_1=c_1(\vec{y}_1)}\ t_1\ \cdots\ \xrightarrow{x_1=c_n(\vec{y}_n)}\ t_n\ \right]\!\!\right]_E =$$

let $f \in F$
$F = F \setminus \{f\}$
$\{x_1,\ldots,x_m\} = \mathsf{Var}(t)$
$E = E \cup \{\delta \mapsto f\}$
in

$$\lambda\kappa.\left(\begin{array}{l}[\![\ t_1\ ]\!]_E\ (\lambda t_1.{\ulcorner} f(c_1(\vec{y}_1),x_2,\ldots,x_n) = {\urcorner} +\!\!+ t_1) +\!\!+ \\ \ldots \\ [\![\ t_n\ ]\!]_E\ (\lambda t_n.{\ulcorner} f(c_n(\vec{y}_1),x_2,\ldots,x_n) = {\urcorner} +\!\!+ t_n) +\!\!+ \\ \kappa({\ulcorner} f(x_1,\ldots,x_m){\urcorner})\end{array}\right)$$

$$\left[\!\!\left[\ \delta :\ \boxed{t}\ \xrightarrow{a=a'}\ t_1\ \xrightarrow{a\neq a'}\ t_2\ \right]\!\!\right]_E =$$

let $f \in F$
$F = F \setminus \{f\}$
$\{x_1,\ldots,x_m\} = \mathsf{Var}(t)$
$E = E \cup \{\delta \mapsto f\}$
in

$$\lambda\kappa.\left(\begin{array}{l}[\![\ t_1\ ]\!]_E\ ([\![\ t_2\ ]\!]_E\ (\lambda t_2 t_1.{\ulcorner}\textbf{if } a{=}{=}a \textbf{ then}{\urcorner} +\!\!+ t_1 +\!\!+ {\ulcorner}\textbf{else}{\urcorner} +\!\!+ t_2)) +\!\!+ \\ \kappa({\ulcorner} f(x_1,\ldots,x_m){\urcorner})\end{array}\right)$$

Figure 8.1: Code generation – part I.

$$\left[\!\left[\begin{array}{c} c(t_1,\ldots,t_n) \\ c1 \swarrow \quad \searrow cn \\ t_1 \quad \ldots \quad t_n \end{array} \right]\!\right]_E =$$

$$\lambda\kappa.\left(\left[\!\left[t_1 \right]\!\right]_E (\ldots(\left[\!\left[t_n \right]\!\right]_E (\lambda t_n \ldots t_1.\kappa^{\ulcorner} c(t_1,\ldots,t_n)^{\urcorner}))\ldots)\right)$$

$$\left[\!\left[\delta : \begin{array}{c} t \\ \downarrow \lambda t.\mathrm{TRUE} \\ t' \end{array} \right]\!\right]_E = \begin{array}{ll} \textbf{let} & f \in F \\ & F = F \setminus \{f\} \\ & \{x_1,\ldots,x_m\} = \mathsf{Var}(t) \\ & E = E \cup \{\delta \mapsto f\} \\ \textbf{in} & \\ & \lambda\kappa.\left(\left[\!\left[t' \right]\!\right]_E (\lambda t'.{}^{\ulcorner} f(x_1,\ldots,x_m) =^{\urcorner} + t') + \kappa^{\ulcorner} f(x_1,\ldots,x_m)^{\urcorner}\right) \end{array}$$

$$\left[\!\left[\delta : x \right]\!\right]_E = \lambda\kappa.\kappa^{\ulcorner} x^{\urcorner}$$

$$\left[\!\left[\delta : t \right]\!\right]_E = \begin{array}{ll} \textbf{let} & \gamma \text{ contain the renaming of } t \\ & f = E(\gamma) \\ & \{x_1,\ldots,x_m\} = \mathsf{Var}(t) \\ \textbf{in} & \\ & \lambda\kappa.\kappa^{\ulcorner} f(x_1,\ldots,x_m)^{\urcorner} \end{array}$$

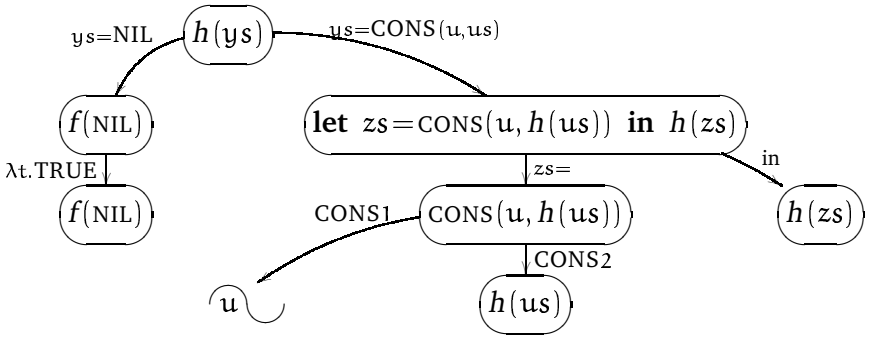
Figure 8.2: Code generation – part II.

Notice how a let-term is transformed into a function call. This is done to ensure that computations from the let-bound variables are not duplicated (which of course only has effect when the intended call-by-need semantics is used).

Example 124
Consider the silly program

$$\begin{array}{lcl}
\textbf{data}\ \mathrm{List}(a) & = & \mathrm{CONS}(a, \mathrm{List}(a)) \mid \mathrm{NIL} \\
f(x) & = & g(x) \\
g(x) & = & f(x) \\
h(\mathrm{NIL}) & = & f(\mathrm{NIL}) \\
h(\mathrm{CONS}(x, xs)) & = & h(\mathrm{CONS}(x, h(xs)))
\end{array}$$

The final process tree for the term $h(ys)$ is



From this process graph we will generate the following program and term:

$$\begin{array}{lcl}
b() & = & b() \\
c(xs) & = & a(xs) \\
a(\mathrm{NIL}) & = & b() \\
a(\mathrm{CONS}(u, us)) & = & c(\mathrm{CONS}(u, a(us))) \\
\leftarrow a(ys) & &
\end{array}$$

The unnecessary function c is a result from ensuring that no duplication of calculation takes place.

Because of the compact nature of the produced process trees, we can generate code from these trees in a very straightforward manner. This is in sharp contrast to previous formulations of supercompilers, where code-generation is intermixed with transition compression. In fact, as noted in [24], the actual optimisation or specialization of the subject program happens during such a post-processing phase. We do not need to do any post-unfoldings, since non-branching nodes in our process trees either stem from non-terminating-function definitions or from the desire to avoid duplication of computation from let-terms. The latter case could be avoided if the code-generating function counted the number of occurrences of let-bound variables in the head of let-terms: if every let-bound variable occurs exactly once, there is no need for generating an extra function call.

Chapter 9

Conclusion and Related Work

We have presented an on-line program transformation technique that transforms a program into an equivalent, but hopefully more efficient program. This process involves aggressively unfolding the program and propagation of information by means of a restriction system. It is shown that the program transformer terminates on all input, which could make it a candidate for inclusion in an optimising compiler.

One of the achievements of this thesis is that we have been able to incorporate propagation of complex negative information into a supercompilation algorithm. Some implementaions of supercompilers for the REFAL language also propagates negative information, but this is done in a very ad hoc manner, and is not described in literature. To our knowledge, non of them are guaranteed to terminate.

Glück and Klimov [10] have presented a supercompiler that propagates negative information, but their language is a bit more simple than the one presented here; their S-graph language works on lisp-like structures and their comparison operation `(EQA? x y)` is limited to comparison of atoms. For this reason, their restriction system can be represented as two lists: one for positive information, *e.g.* $[x \mapsto A, y \mapsto z, u \mapsto \text{CONS}(B, x)]$, and one for negative information, *e.g.* $[x \neq \text{CONS}, A \neq z]$. They show that this is in fact enough to pass the KMP-test, but there is no termination guarantee for their transformer.

For logic programs, Leuschel and De Schreye have presented a transformation technique called Constrained Partial Deduction [19], from which we have adopted the terminology of local and global trees. The aim of partial deduction is to specialize a goal which consists of a set of atoms. The *global control* decides which atoms should be partially deduced. The *local control* construct a fintite SLDNF-tree for each atom selected, thus producing new atoms. The local control must then ensure that the SLDNF-trees are indeed finite, and the global control must ensure that only a finite number of atoms are produced. It might then be necessary to collapse several atoms into one by means of generalisation, which is *constrained* by the structure of atoms that take part in the generalisation. Their constraints are thus used to increase the precision of generalisations.

For rewrite-based functional-logic programs, Lafave and Gallagher have develop a partial evaluator that propagates sets of constraints during partial evaluation. The constraints are, like ours, used to decide the outcome tests in the program. Their transformation performs a series of local rewrites which corresponds to our local trees. Like in our supercompiler, the homeomorphic embedding is used to ensure termination of local unfoldings and generalisation is employed to ensure global termination, but this is not proved. A non-specified constraint solver is assumed to be present.

The idea of employing a special kind of machinery to prune infeasible branches is also pre-

sented by Takano [28], but in this paper the machinery is an unspecified automated theorem prover. He reports to have achieved automatic specialization of a naïve string matcher such that a program that is as efficient as the KMP-matcher is obtained.

Appendix A

Homeomorphic Embedding

Let $P, Q \in T(\mathscr{C} \cup \mathscr{F} \cup \{\textbf{ifthenelse}\}, \mathscr{X})$. Finding out whether $P \trianglelefteq Q$ can be done in $O(|P| \cdot |Q|)$ sketched as follows.

Enumerate the subtrees in P breadth first such that $\mathit{number}(\epsilon) = 0$ and siblings have consecutive numbers from left to right. For all symbols $e \in \mathscr{C} \cup \mathscr{F} \cup \{\textbf{ifthenelse}\}$, calculate a function $\mathit{constructors} : \mathcal{E} \to [\mathbb{N} \times \mathbb{N}]$ on the basis of the structure of P: for each symbol e, *constructors* maps e to a list of pairs $[\zeta_1, \ldots, \zeta_n]$, where, for each pair ζ_i, the second component $\pi_2(\zeta_i)$ is the number of some node labelled by e and the first component $\pi_1(\zeta_i)$ is the number the leftmost child of the second component, *i.e.* when a node δ in P has $P(\delta) = e$, then $\mathit{constructors}(e) \ni (\mathit{number}(\mathit{leftmost}(\delta)), \mathit{number}(\delta))$. This can be done in $O(|P|)$ time.

With this mapping at hand, we can traverse the tree Q thus: For each node δ in Q, calculate a list ℓ_δ of subtrees in P that are embedded in the subtree $Q\lceil\delta\rceil$ rooted in δ recursively as follows: let $Q(\delta) = e$ and $\gamma_1, \ldots, \gamma_n$ be the children of δ, and let $\{\ell_1, \ldots, \ell_n\}$ be the lists of embedding for $\gamma_1, \ldots, \gamma_n$, respectively.

1. Let the list h_δ of newly discovered embedding by empty. For each pair (m, p) in the list of pairs $\mathit{constructors}(e)$, traverse the list ℓ_1 until $\mathit{head}(\ell_1) \geq m$.

   (a) If $\mathit{head}(\ell_1) = m$, move on to ℓ_2 and search for $m+1$ (since siblings are numbered consecutively), and so forth. If all the lists ℓ_i from the childen were succesfully matched in this way, it means that we have an embedding of the subtree numbered p in δ, and thus we put p into h_δ.

   (b) If at some point $\mathit{head}(\ell_{\gamma_i}) > (m + i - 1)$, it means that the subtree numbered p is *not* embedded in δ. Hence we do not add anything to h_δ.

   When the list $\mathit{constructors}(e)$ has been traversed in this way, the list h_δ will contain all *new* embeddings for the subtree δ. This will take in $O(|P|)$ time. Of course, all the embeddings for the children of δ (which are ℓ_{γ_i}) are also embeddings for δ, so the children's lists and the list of new embeddings has to be unioned, and the result will be the full list ℓ_δ of embeddings for δ. This can be done in $O(|P|)$ time, since the number of children are bounded by the maximal degree of the constructors/functions.

Algorithm 125 (Homeomorphic embedding)
Let $\mathcal{E}_q$ be a bounded set of symbols, $\mathscr{X}$ be an infinite number of variables and let $P, Q \in T(\mathcal{E}_q, \mathscr{X})$. The homeomorphic embedding ($P \trianglelefteq Q$) can be calculated by the program in figures A.1 and A.2. The functions "map" and "reverse" are standard list functions. The function "queue" turns a list into a standard $O(n)$ amortised time queue.

```
input P, Q, E_q
let constructors(e) = [] for all e ∈ E_q
let variables= []
let worklist= queue([(ε, 0)])
let m = 1
while worklist is not empty
  let work= head(worklist)
  let δ = π1(work)
  let n = π2(work)
  let e = P(δ)
  if δ ∈ leaf(P)
    if e ∈ X
      variables = n:variables
    else
      constructors(e) = (0, n):constructors(e)
  else
    constructors(e) = (m, n):constructors(e)
    foreach γ ∈ children(P, δ)
      worklist = worklist ++ (γ, m)
      m = m + 1
endwhile
constructors(e) = reverse(constructors(e)) for all e ∈ E \ X
variables = reverse(variables)
let ℓ = embeddings(constructors, variables, Q, ε)
if head(ℓ) = 0 then return true else return false
```

Figure A.1: Homeomorhic embedding algorithm (Preprocessing of the tree P).

```
embeddings(constructors, variables, Q, δ) =
  let e = Q(δ)
  if δ ∈ leaf(Q)
    if e ∈ 𝒳
      return variables
    else
      return (map π2 constructors(e))
  else
    let {γ1, ..., γn} = children(Q, δ)
    let ℓi =embeddings(constructors, variables, Q, γi) for all i
    let ℓ'i = ℓi for all i
    let ℓ = []
    let ℏ = constructors(e)
    while ℏ is not empty
      let min= π1(head(ℏ))
      let i = 1
      while i ≤ n and ℓ'i is not empty
        while ℓ'i is not empty and head(ℓ'i) < min
          ℓ'i = tail(ℓ'i)
        if ℓ'i is not empty and head(ℓ'i) = min
          min = min + 1
          i = i + 1
      endwhile
      if i = n + 1
        ℓ = π2(head(ℏ)):ℓ
      ℏ = tail(ℏ)
    endwhile
  endwhile
  return reverse(ℓ) ∪ ℓ1 ∪ ... ∪ ℓn
```

Figure A.2: Homeomorhic embedding algorithm (Recursively calculate all embeddings).

Bibliography

[1] AUGUSTSSON, L. Compiling lazy pattern matching. In *Conference on Functional Programming and Computer Architecture* (1985), J.-P. Jouannaud, Ed., vol. 201 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 368–381.

[2] AUGUSTSSON, L. Partial evaluation in aircraft crew planning. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-97)* (New York, June12–13 1997), vol. 32, 12 of *ACM SIGPLAN Notices*, ACM Press, pp. 127–136.

[3] BURSTALL, R., AND DARLINGTON, J. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery 24*, 1 (1977), 44–67.

[4] COMON, H., AND LESCANNE, P. Equational problems and disunification. *Journal of Symbolic Computation 7*, 3–4 (Mar.–Apr. 1989), 371–425.

[5] COURCELLE, B. Fundamental properties of infinite trees. *Theoretical Computer Science 25* (1983), 95–169.

[6] DAMAS, L., AND MILNER, R. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages* (Jan. 1982), pp. 207–212.

[7] DANVY, O., GLÜCK, R., AND THIEMANN, P., Eds. *Partial Evaluation*, vol. 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[8] DERSHOWITZ, N. Orderings for term-rewriting systems. *Theoretical Computer Science 17* (1982).

[9] FUTAMURA, Y. Partial evaluation of computing process – an approach to a compiler-compiler. *Systems, Computers, Controls 2*, 5 (1971), 45–50.

[10] GLÜCK, R., AND KLIMOV, A. V. Occam's razor in metacomputation: the notion of a perfect process tree. In *Static Analysis. Proceedings* (1993), G. Filè, P.Cousot, M.Falaschi, and A. Rauzy, Eds., vol. 724 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 112–123.

[11] GLÜCK, R., AND SØRENSEN, M. H. A roadmap to metacomputation by supercompilation. In *Partial Evaluation* (1996), O. Danvy, R. Glück, and P. Thiemann, Eds., vol. 1110 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 137–160.

[12] JOHNSSON, T. Lambda lifting: Transforming programs to recursive equations. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (Nancy, France, Sept. 1985), J.-P. Jouannaud, Ed., vol. 201 of *LNCS*, Springer, pp. 190–203.

[13] JONES, N. D., GOMARD, C. K., AND SESTOFT, P. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[14] KIRCHNER, C., AND LESCANNE, P. Solving disequations. In *Proceedings, Symposium on Logic in Computer Science* (Ithaca, New York, 22–25 June 1987), The Computer Society of the IEEE, pp. 347–352.

[15] KLOP, J. W. Term rewriting systems. In *Handbook of Logic in Computer Science*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds., vol. 2. Oxford University Press, Oxford, 1992, ch. 1, pp. 1–117.

[16] KNUTH, D., MORRIS, J., AND PRATT, V. Fast pattern matching in strings. *SIAM Journal on Computing 6*, 2 (1977), 323–350.

[17] LASSEZ, J.-L., MAHER, M., AND MARRIOTT, K. Unification revisited. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufmann, Los Altos, Ca., 1988, pp. 587–625.

[18] LAUNCHBURY, J. A natural semantics for lazy evaluation. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Charleston, South Carolina, January 10–13, 1992* (New York, NY, USA, 1993), ACM, Ed., ACM Press, pp. 144–154.

[19] LEUSCHEL, M., AND DE SCHREYE, D. Constrained partial deduction and the preservation of characteristic trees. *New Generation Computing* (1997). To Appear.

[20] MILNER, R. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17* (1978), 348–375.

[21] PETTOROSSI, A., AND PROIETTI, M. A comparative revisitation of some program transformation techniques. In Danvy et al. [7], pp. 355–385.

[22] PLASMEIJER, R., AND VAN EEKELEN, M. *Functional Programming and Parallel Graph Rewriting*. Addison Wesley, 1993.

[23] SIMONYI, C. The death of computer languages, the birth of intentional programming. Tech. Rep. MSR-TR-95-52, Microsoft Research, 1995.

[24] SØRENSEN, M. Turchin's supercompiler revisited. Master's thesis, Department of Computer Science, University of Copenhagen, 1994. DIKU-rapport 94/17.

[25] SØRENSEN, M. Convergence of program transformers in the metric space of trees. In *Mathematics of Program Construction* (1998), J. Jeuring, Ed., vol. 1422 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 315–337.

[26] SØRENSEN, M., AND GLÜCK, R. Introduction to supercompilation. In *DIKU Summer school on Partial Evaluation* (1999), Lecture Notes in Computer Science, Springer-Verlag. To appear.

[27] SØRENSEN, M. H., GLÜCK, R., AND JONES, N. D. A positive supercompiler. *Journal of Functional Programming 6*, 6 (1996), 811–838.

[28] TAKANO, A. Generalized partial computation using disunification to solve constraints. In *Conditional Term Rewriting Systems, CTRS-92, Pont-à-Mousson, France, July 1992 (Lecture Notes in Computer Science, vol. 656)* (1993), M. Rusinowitch and J. L. Rémy, Eds., Berlin: Springer-Verlag, pp. 424–428.

[29] TURCHIN, V. A supercompiler system based on the language Refal. *SIGPLAN Notices 14*, 2 (1979), 46–54.

[30] TURCHIN, V. The language Refal, the theory of compilation and metasystem analysis. Courant Computer Science Report 20, Courant Institute of Mathematical Sciences, New York University, 1980.

[31] TURCHIN, V. The use of metasystem transition in theorem proving and program optimization. In *Automata, Languages and Programming* (Noordwijkerhout, Netherlands, 1980), J. de Bakker and J. v. Leeuwen, Eds., vol. 85 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 645–657.

[32] TURCHIN, V. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst. 8*, 3 (1986), 292–325.

[33] TURCHIN, V. Program transformation by supercompilation. In *Programs as Data Objects* (1986), H. Ganzinger and N. D. Jones, Eds., vol. 217 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 257–281.

[34] TURCHIN, V. Metacomputation: Metasystem transition plus Supercompilation. In Danvy et al. [7], pp. 481–510.

[35] TURCHIN, V., NIRENBERG, R., AND TURCHIN, D. Experiments with a supercompiler. In *ACM Conference on Lisp and Functional Programming* (1982), ACM Press, pp. 47–55.

[36] WADLER, P. Efficient compilation of pattern matching. In *The Implementation of Functional Programming Languages*, S. Peyton-Jones, Ed. Prentice-Hall, 1987.

[37] WADLER, P. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science 73* (1990), 231–248.

Index