

Automating Functional Program Transformation

Markus Mottl



MSc in Artificial Intelligence — Foundations of AI
Division of Informatics
University of Edinburgh
2000

Abstract

We present a framework for automatic program transformation of a strict and pure functional language with a well-defined semantics. It will be shown that such a framework can be implemented most declaratively and concisely in a recently developed higher-order logic programming language called LambdaProlog.

The most important component of this framework is an efficient, always terminating partial evaluator that can handle higher-order functions and preserves the effect behaviour of programs by making use of monads, a construct which originated in category theory. This allows us to fully exploit the higher-order capabilities of the implementation language to reason about computations as required for partial evaluation. Due to this technique, the only factor that limits the optimisation power of the partial evaluator seems to be the generally undecidable problem of inferring termination behaviour of computations. The information gathered by the partial evaluator is most useful for subsequent improvements such as, for example, common sub-expression elimination.

We will also give a broad overview of techniques to automate program transformation, how their correct application can be guaranteed and how such transformation processes can be guided to quickly find more efficient programs. The framework should provide for a strong basis to try out some of the more advanced transformations.

Acknowledgements

First of all, I would like to thank my first supervisor, Dr. Alan Smaill, for his encouragement to attempt this project and his positive criticism that kept me on track. I also owe much to my second supervisor, Dr. Ewen Denney, who pointed me to many details I was not aware of. It must have been hard for them to endure a student who worked on the automated transformation of strict functional programs with a lazy attitude.

A thousand words would not be enough to thank my mother for her continuous moral and financial support: without her my studies would have been impossible.

If it were not for Dr. Andreas Geyer-Schulz, my former supervisor at the Vienna University of Economics, I would have never discovered the mysterious field of Artificial Intelligence. Thanks a lot!

I owe at least half of my knowledge on both theory and practice of functional programming to the members of the Caml mailing list and the Usenet group “comp.lang.functional”: thanks for being an infinite source of wisdom!

Many thanks, too, to all my colleagues on the course and to the “Edinburgh University Renaissance Singers”: their welcome distraction showed me that there is life outside of computer labs.

Though I have never enjoyed meeting him personally, I am deeply indebted to Wolfgang Amadeus Mozart: his music brought light even into my darkest hours.

Last but not least, I am grateful to the SAAS, the Student Awards Agency for Scotland, which supported my studies in Edinburgh by paying my tuition fees.

Contents

1	Introduction	1
1.1	Purpose of program transformations	3
1.2	Why transform functional programs?	4
1.3	Difficulties	7
1.3.1	Semantic complications	7
1.3.2	Complexity of search	9
1.4	Goals and structure of this work	11
2	A Simple Functional Language with Call-by-Value Semantics	13
2.1	Description of types, values and informal semantics	14
2.1.1	Unit type	15
2.1.2	Product type	16
2.1.3	Sum type	16
2.1.4	Function type	18
2.1.5	Type of recursively defined functions	18
2.1.6	Recursive types	20
2.2	Abstract syntax	22
2.3	Formal typing rules	23
2.4	Operational semantics	24
2.4.1	Values	24
2.4.2	Structural operational semantics	26
2.5	Denotational semantics	28
3	Automated Transformations	31
3.1	Partial evaluation	32
3.1.1	Example	32
3.1.2	Properties of partial evaluation	34
3.2	Fold/Unfold method	35
3.2.1	Deforestation	40
3.2.2	Tupling transformation	41
3.3	Bird/Meertens formalism	43

4	Correctness of Transformations	45
4.1	Preserving termination behaviour	46
4.1.1	Correctness of partial evaluation	47
4.1.2	Correctness of Fold/Unfold transformations	48
4.2	Other side effects	53
4.3	Practical correctness issues	55
5	Controlling Transformations	59
5.1	Reducing the number of choice points	60
5.1.1	Partial evaluation	60
5.1.2	An advanced technique for guiding search — rippling	62
5.2	Giving priority to choices	63
5.3	Control and completeness	64
5.4	Examples of control heuristics	64
5.4.1	A classic control strategy	64
5.4.2	A control strategy with partial evaluation	65
6	Implementing Transformations	67
6.1	Implementation in LambdaProlog	67
6.1.1	Higher-order logic	69
6.2	Structuring partial evaluation using monads	72
6.2.1	What are monads?	74
6.2.2	Using monads to hide side effects	75
6.2.3	Partial evaluation and monads	76
6.2.4	Extensibility and monads	79
6.2.5	Theoretical aspects of monads and partial evaluation	80
6.2.6	Final remarks on the use of monads	81
6.3	Maintaining types during partial evaluation	82
7	System Evaluation	83
7.1	Type inference	83
7.2	Evaluation	84
7.3	Partial evaluation	85
7.4	Common sub-expression elimination	90
7.5	Inlining	91
8	Conclusion	93
8.1	A better partial evaluator?	93
8.2	Correctness issues	95
8.3	Control issues	96
8.4	New ways of implementation	96

8.5	Future work	97
8.6	Completely different ways	97
Appendices		102
A	Miscellaneous Examples	103
A.1	Fibonacci in $O(\log_n)$ — Haskell code	103
B	Components of the system	105
C	Examples of System Code	109
C.1	Code of effect monad	110
C.1.1	Signature of effect monad	110
C.1.2	Implementation of effect monad	111
C.2	Code of monadic partial evaluator	112
C.2.1	Signature of monadic partial evaluator	112
C.2.2	Implementation of monadic partial evaluator	112

List of Tables

2.1	Abstract syntax	22
2.2	Formal typing rules	23
2.3	Values	25
2.4	Structural operational semantics	27

Chapter 1

Introduction

Many years ago I watched a famous ventriloquist as he “talked” to his puppet. “I can calculate faster than anybody else in the world!”, it said. “Well, then how much is five times five?”, asked the master. Blindingly fast the figure answered: “23!”. The master complains: “But that was wrong!”. The puppet replies in a superior tone: “Yes, but fast...”.

Somewhat subconsciously we accept the fact that correctness and efficiency of computation are two conflicting goals. The tension between them has never been as obvious as in our time, where the correct and efficient handling of information, of doing computation, has become crucial for everyday life.

If we want to book a flight, we expect that information displayed by a booking computer is correct and that we receive the answer immediately. The engineer who simulated the aerodynamic behaviour of our plane on a computer must have been able to trust the results for the sake of our safety, while his company might have had interest in efficient computer programs for this purpose to reduce development time and costs.

Unfortunately, experience shows that we often do a bad job in implementing information systems. Sometimes they just fail to do their job correctly, which can range from e.g. exploding space rockets (Ariane 5) to crashing

of widespread operating systems for the average home user. On the other hand, inefficient computer programs can cause international companies to go bankrupt, as, for example, was the case for a large vendor of pharmaceutical products, who sued a consulting company for having installed a standard enterprise solution that just could not cope with their high data traffic.

How does it happen that it is so difficult to find both correct and efficient solutions to our problems?

If there are several correct solutions to a problem, it is a natural assumption that the simplest one will be found first, or that there is at least a bias towards finding simpler solutions before complex ones. Though the simplest one may indeed be most efficient, it is much more likely that there are more efficient complex ones — simply because complexity means that we have to choose them from a much larger set, which makes it more probable to find efficient solutions there.

The reason is that given no information at all, which element from a set (of solutions) has a specific property (it is the “best”), we have to assign equal probabilities to them so as not to introduce an unjustified bias.

Choosing possibly more sophisticated solutions from a large space is not only more work. In addition, it raises the probability that we accept an incorrect one: “To human is err.”

A great deal of time has been invested in the development of tools that support us in finding solutions or in improving already existing ones automatically. Modern approaches generally use the powerful concept of formal methods to provide for full automation of problem solving. This usually requires formalising the problem domain using *formal languages*, which results in symbolic representations that are suitable for manipulation by machines.

In fact, modern universal programming languages belong in this group of formal languages, and they claim to be suitable for formalising all computable tasks and functions in general. If we can use them to solve our practical

problems, then it is an obvious step to apply their power to the problem at hand: we write programs to develop efficient programs!

Taking up the idea of automating generation of efficient programs, there are basically two applicable techniques. We can try to:

- generate efficient programs from a problem specification by using automated *program synthesis*.
- use existing programs as starting point to finding better ones through the use of automated *program transformation*.

The first task is surely the more challenging one. A system capable of it would not have any hint at all where to start searching. While the second technique looks less powerful, it is still of great interest to the software industry: human programmers often find it nearly as easy (or should we say: difficult?) to implement simple, though possibly inefficient programs as writing correct formal problem specifications in very abstract specification languages. Taking up a human's initial hint, a transformation system could arrive at good solutions much faster. In our work we will focus on this second technique.

1.1 Purpose of program transformations

What exactly is a program transformation system supposed to do? Although producing more efficient versions of given programs is most likely the primary intention of most users, the most important aspect to the developer of the transformation system itself is something else: correctness. Because it can be extremely difficult to verify the result of the transformation process, we have to design the system in such a way that the meaning of transformed programs is provably the expected one.

It actually only rarely happens that transformations are used to change the meaning of programs, for example to correct previous misbehaviour. This was more frequently the case during the transition period before the year 2000: some companies developed automated tools to transform programs containing insidious Y2K-bugs — with varying success. Though the behaviour of these programs was correct up to this time, encoding years using two digits only would have lead to incorrect handling of dates after the change to the new century. Therefore, the (wrong) meaning of the programs had to be changed by transformation to cover the more general case.

Other transformations involve simplification of programs or bringing them into a canonical form which might be easier to analyse. However, the main intention is most often optimisation: the user can focus on quickly and correctly solving the problem and leaves efficiency considerations mostly to the transformation system.

Before everything else, the purpose of optimising transformations is to preserve the meaning of programs¹: otherwise the result would be potentially unusable for any purpose. Additionally, it would be very surprising for the user if his program ran slower rather than faster after the transformation, which should be avoided, too.

1.2 Why transform functional programs?

So far, we have not justified the choice to transform functional programs as indicated in the title of the thesis. In fact, functional programming is only one of several different approaches and by far not the most widespread programming paradigm.

¹This issue will be discussed in more detail in chapter 4.

Logic Programming The most general and convenient way to use machines for solving problems using formal languages would be to just tell them, *what* problem should be solved. We call this way of implementing programs *declarative programming*. One of its variants, *logic programming*, is a very general method to achieve this high level of automated problem solving.

This style abstracts from the need to know *how* to find solutions (control knowledge). Unfortunately, the software industry has a certain reservation against such high level languages. They are often considered as “too mathematical”, “too abstract” or “too inefficient”, which is generally not justified: if we cannot even specify *what* problem we want to solve, how can we know *how* to solve it?

The conciseness of declarative programs often exhibits how difficult the problem to be solved really is, whereas less abstract languages hide the problem complexity in much larger amounts of more concrete code (i.e. each single step can be followed more easily). This may be an explanation for psychological inhibitions against declarative languages. Efficiency of logic languages has vastly improved over the last decades, modern implementations being competitive with low-level languages. This should be a strong enough argument against efficiency concerns.

Examples of logic languages are (Lambda)Prolog, Mercury and Oz. The language we will use to implement some example transformations is Lambda-Prolog. We will explain in chapter 6 which of its features make this a viable choice.

Imperative Programming The prevalent programming paradigm, which is strongly concerned about the “how”, is *imperative programming*, where the user tells the computer each action to be done after another. Due to the von Neumann architecture of most modern computers, which lends itself to this style, these languages are considered as leading to most efficient programs.

However, knowing “how” to do something is generally much more difficult than knowing “what” to do. Thus, this style is in practice tedious, error-prone and among the main reasons for failures of large software systems. Widespread languages of this kind are C/C++, Java, Fortran, Basic and Cobol.

Functional Programming One feature that imperative programs lack is that they cannot be easily transformed into equivalent ones. The reason is that the meaning of imperative constructs heavily depends on the state of computation. But the state is only exactly known at runtime, which makes it nearly impossible to apply equational reasoning to parts of programs. The style which addresses this problem is *functional programming*. Here, a program is treated as a function, which is usually composed out of other functions. Higher-order functions, ones that can be passed as argument or returned as result, are also an important aspect of functional programming and allow implementation of very generic solutions. Type systems of functional languages are generally very powerful and significantly more advanced (securer, more flexible) than those of their imperative competitors.

Functions are a very basic and well understood mathematical concept, and equational reasoning can be applied to them easily. This means that transformation systems can build on a sound basis. This elegant feature also helps humans to reduce the complexity of programs: the evaluation of different parts of the program cannot change the meaning of other parts. If something does not work, the offending part can in most cases be pinpointed much faster. Well-known languages include Haskell and Clean, SML and OCaml, Lisp and Scheme.

Functional programming is usually also considered as declarative programming due to its strong mathematical foundations, even if logic programming surpasses it in generality (functions are a special case of relations

(predicates), the latter being the core elements of logic programs). Still, functions have properties that make them especially amenable to transformations that build on equational reasoning.

It is the functional style on which our transformations will focus, and we will explain the semantics of a language of this kind in more detail in chapter 2.

1.3 Difficulties

1.3.1 Semantic complications

As indicated in the last section, functional languages support transformations in a straightforward manner: we can, for example, derive new programs by substituting parameters of function applications in their corresponding function bodies — almost. One problem that appears here is that the correctness of this very important transformation depends on the evaluation strategy of the language. As we will see later, it is of great necessity to specify an unambiguous formal semantics for the language to guarantee correct transformations.

Call-by-name semantics

Most transformation techniques presented in the literature assume free substitutability, which is only possible in functional languages with so-called *call-by-name* evaluation: this does not evaluate function arguments before applying them to the function, but substitutes the whole unevaluated expression in the function body. This allows terminating evaluation of a larger class of programs.

Although several languages are defined in terms of such a semantics (e.g. Haskell and Clean), there are also some disadvantages associated with this

approach:

- Compiler implementors find it generally much harder to write efficient code generators for such languages.
- Call-by-name evaluation (which is often implemented as *lazy evaluation*) requires *purity* so as not to make understanding of program execution close to impossible: this means that these languages completely forbid free use of imperative elements, not only destructive assignment to variables, but also input/output functions. Several techniques have been proposed to lift this restriction (e.g. uniqueness types in Clean; monads in Haskell). Still, beginners, especially ones who are used to imperative programming, find it difficult to grasp these concepts easily.
- Although understanding the meaning of programs is fairly straightforward in pure and lazy functional languages, other important program properties are not necessarily easy to reason about: it is generally very difficult to establish worst-case bounds on time and especially memory consumption of programs in such languages².

Call-by-value semantics

The alternative to the semantics above is *call-by-value*, sometimes referred to as *strict evaluation*: here, function parameters are always evaluated before they are substituted within the function body. This is usually more efficient; it may, however, lead to non-termination in cases, where call-by-name semantics would yield a result. Otherwise, reasoning about runtime properties becomes tractable, and languages of this type (e.g. OCaml and SML) can more easily have coexisting pure (purely functional) and impure (imperative) features, which may make them a better choice for beginners.

²See [Oka98] for further information on advanced implementation of and reasoning about purely functional programs.

Due to the mentioned shortcomings of call-by-name evaluation, which may be lifted in the future by more research, and because there has not yet been so much work on transformations of strict languages, we will restrict this work to the latter evaluation strategy. This comes at the expense of losing the full power of equational reasoning due to possible impurities and non-termination. We will see in this work that techniques used in purely functional languages to tame impurity of side-effecting (= imperative) code can be very valuable to achieve our goals.

1.3.2 Complexity of search

Besides semantics related problems that complicate the correct application of program transformations and impact effectiveness, an equally important question concerns the efficiency of transformation systems. The complexity of search for more efficient programs depends on several parameters:

- the size of the input program,
- the number of transformation rules,
- the number of valid transformations applicable to a given program at a specific stage of the transformation process.

Both the first and especially the second parameter can lead to exponential explosion of the size of the search space. To give an example, a program may contain large mathematical expressions. If we try to simplify them by applying transformation rules that exploit mathematical properties like commutativity and associativity in all possible ways, this may require a significant effort.

There is naturally not much we can do to limit the impact of the first parameter other than not transforming parts of the program. Limiting the influence of the second one by removing rules usually makes the system less

powerful and may not allow exploitation of all opportunities to improve programs.

The last parameter, however, gives us some control over the search process: if we find out in which cases specific transformations are more suitable than others, maybe even a “sure bet”, then we can structure their application in such a way that we always only consider the most promising ones. Such techniques of reducing the number of choices (the branching factor of the search space) are called *heuristics*.

We will see that it is necessary to employ heuristics to make search for more efficient programs tractable. *Partial evaluation*, a technique that can statically evaluate programs (without knowing their input), will play a major role here to reduce the number of alternatives in the search space³.

It is important to point out that the correctness of the transformations is independent of the heuristic: the intention behind heuristics is only to restrict the number of alternatives to the most promising ones or to impose an order on them in which they are tried. They do not add new (potentially unsound) choices.

As is the case for rewrite systems in general, sets of program transformation rules do not necessarily always allow normal forms: this means that there can be programs such that the transformation process does not terminate in a state in which no transformation rule applies to the program. On the other hand, restricting the number of rule applications to prevent this problem can result in a loss of completeness: it may still be possible that the program could be improved by further transformations.

It is often the case that such situations arise in program transformation systems. The consequences of this problem can be weakened by employing search strategies that can be parameterised in terms of the search depth. This allows the user of such systems to decide, on a per-case basis, how much

³This will be discussed in more detail in chapter 4.

computation time they are willing to sacrifice to achieve a higher degree of completeness with deeper search.

1.4 Goals and structure of this work

The aim of the project is to implement a basic framework in which various transformations can be tried out easily on a language which has a rigorously defined semantics.

The framework contains functionality for handling abstract syntax trees of the language, inferring types (including type checking of polymorphically typed programs), termination analysis, common sub-expression elimination and, most important, a fairly general partial evaluator that can handle higher-order functions while exactly preserving the meaning of programs, including termination behaviour.

The reader might especially benefit from taking a look at the approach taken in the partial evaluator⁴. It is implemented in monadic style, which makes the program very declarative in nature: short, clear, and easily extensible. It seems that this approach can be used to conveniently structure transformation systems in the general case. Its implementation relies heavily on higher-order features of the relatively young logic programming language *LambdaProlog*. The thesis will also outline ways to continue implementation of transformations in this specific framework.

In the chapters to follow we will deal with the following aspects of automating functional program transformation:

- We will specify the formal semantics of a simple, strict and pure functional language in chapter 2.
- Using this language, we will provide a short overview of the state of

⁴See section 6.2 for more information.

research concerning transformation of functional programs: chapter 3.

- The importance of the topic demands a detailed treatment of how to ensure correctness of various transformations and of associated practical considerations that may be important to remember: chapter 4. This work contributes to solving problems of correctness associated with preserving termination behaviour and other effects during partial evaluation.
- We will show how search for efficient programs can be controlled by introducing alternating stages of transformation: partial evaluation and more sophisticated transformations (e.g. folding) follow each other in turn to produce more efficient programs. This is discussed in chapter chapter 5.
- Implementation of correct, extensible and efficient program transformation systems is difficult. Therefore, a report on the challenges encountered and on techniques developed to overcome them will be given in chapter 6. Chapter 7 will evaluate the features implemented in the system with examples.
- The last part of the thesis, chapter 8, sums up our work and gives final hints that may lead to further extensions and improvements of the system.

Chapter 2

A Simple Functional Language with Call-by-Value Semantics

In this chapter we specify the semantics of the language which we use to implement transformations.

During recent years, several methodologies to rigorously specify the semantics of formal languages have emerged. We will present the semantics of a simple functional language that is suitable for demonstrating in principle all kinds of transformations that can be applied to more fully featured, “real” functional languages.

The concrete syntax, the one in which the programmer writes his programs, will not be considered: it is less important, how programs look than what they mean and how they are structured. Therefore, only the abstract syntax, which carries the semantics, will be presented. Everything else is just a matter of taste.

Due to practical restrictions on the input character set of computer languages, which disallow convenient and concise mathematical symbols in computer programs, we shall explain the commonly known notation for abstract syntax of functional languages as it is used in the text part of the thesis

together with the keywords that appear in the implementation (Lambda-Prolog terms). This is to prevent confusion of people who want to extend the implementation.

Much care was taken to ensure that the implementation of the language stays conformant to the rigorous basis provided in [Win93] (especially chapter 13 on a strict functional language with recursive types). This should make it much easier to prove properties of the language building upon the strong formal foundations provided in the mentioned book¹.

2.1 Description of types, values and informal semantics

Types are sets of values that belong together in a meaningful way. E.g. the set of integer values normally has a corresponding type in most computer languages. More generally, the notion of types extends to terms, whose evaluation may or may not terminate. If a term is regarded to be of type t and if its evaluation terminates, the resulting value must indeed be of the required type. We write $t : \tau$ to indicate that term t is of type τ .

Type checking ensures to a high degree that the programmer does not accidentally write meaningless programs by combining terms that do not belong together. The implementation part of this work contains both a type inference engine and a type checker for polymorphically typed programs of the given language. Polymorphism allows families of types to be parameterised by others. This is a very expressive and generic way to deal with types.

*Type inference*² is a process that can automatically infer which type a program and its constructs have by analysing their structure. Unfortunately, type inference for polymorphically typed languages, which is the case for

¹The lecture notes [Sim99] helped answer some questions, too.

²See [MS95] for an introduction to polymorphic type inference.

many functional ones, is undecidable in the general case. Therefore, if we want to be able to assign correct types to all possible terms, we need type annotations provided by the user that enable the compiler to infer correct types: it would otherwise have to reject type correct programs or its type inference algorithm could loop. Both cases are not very satisfactory.

When an algorithm verifies programs with such annotated types, we speak of *type checking*. The implementation of our system features both a type inference algorithm and an extension for type checking that can be used to verify the type correctness of input programs. Unfortunately, it is still not fully complete: it cannot yet handle recursive types in the general case. The current naive version loops under some circumstances.

The implementation of the type system in our system started out with an example implementation presented in John Hannan’s tutorial [Han98], corrected three small errors, extended the type system to cover sum types and recursive types (see further below) and changed the way recursive functions are handled to a more preferable style for strict functional languages. To conform better to the formal foundation of the example language as presented in [Win93], a few identifiers were renamed, too (tabs \leftrightarrow tlam; abs \rightarrow lam).

The type system of our language is otherwise very simple. In the following section, we will give a detailed specification of the types of our language and, as we go along, also explain the way they are represented in the implementation that is written in LambdaProlog:

2.1.1 Unit type

This type has only a single value:

	Text notation	Representation in implementation
type	<i>unit</i>	unit
value	*	u

Although this type may seem boring, we need it for e.g. lifting terms

to values by creating a function that takes this value of “zero-information” and returns the given term. This is called *thunking*³ and allows passing around computations as values. This can be used to e.g. simulate call-by-name evaluation in a strict language. Additionally, the unit value may be necessary for specifying datastructures together with sum types (see below).

2.1.2 Product type

Values of this type are pairs of the form (c_1, c_2) , where c_1 and c_2 are values of type τ_1 and τ_2 respectively. For terms $t_1 : \tau_1$ and $t_2 : \tau_2$, we have the pair $(t_1, t_2) : \tau_1 * \tau_2$. It is possible to project from pairs:

Given a term:

$$t : \tau_1 * \tau_2$$

We have:

$$\text{fst}(t) : \tau_1$$

$$\text{snd}(t) : \tau_2$$

where $\text{fst}(t)$ returns the first element of the pair, $\text{snd}(t)$ the second one.

	Text notation	Representation in implementation
type	$\tau_1 * \tau_2$	TP1 ** TP2
value	(c_1, c_2)	pair C1 C2
term	$\text{fst}(t)$	fst T
term	$\text{snd}(t)$	snd T

2.1.3 Sum type

Values of this type have either the form $\text{inl}(c_1)$ or $\text{inr}(c_2)$, where c_1 is a value of type τ_1 and c_2 one of type τ_2 . As before, this extends similarly to terms.

³See [HD97b] for details.

Given terms:

$$t : \tau_1 + \tau_2$$
$$t_1 : \tau \text{ possibly containing variable } x : \tau_1$$
$$t_2 : \tau \text{ possibly containing variable } y : \tau_2$$

We have:

$$\text{case } t \text{ of } \text{inl}(x).t_1, \text{ inr}(y).t_2 : \tau$$

Representation in the implementation:

$$\text{case T LF RF}$$

The case statement has the intuitive meaning that if t evaluates to $\text{inl}(x)$, then this term returns t_1 , otherwise if it evaluates to $\text{inr}(y)$, then t_2 is returned.

The LambdaProlog construct demands some explanation⁴: `case` is just a normal user-defined data constructor that takes three parameters. The first parameter, the variable `T`, stands for the term that decides, what case arm should be taken. The other two parameters represent the case arms. They are actually functions in LambdaProlog. E.g. if the variable `T` matches some pattern `inl v`, then the left case arm will be chosen. If the function representing this case arm is e.g. $(\mathbf{x} \backslash \mathbf{x})$, which is the anonymous⁵ identity function in LambdaProlog notation, then `x` will be bound to `v` in the function body. The result of the case statement would be `v` in this example.

	Text notation	Representation in implementation
type	$\tau_1 + \tau_2$	TP1 ++ TP2
value	$\text{inl}(c)$	<code>inl C</code>
value	$\text{inr}(c)$	<code>inr C</code>

⁴We will go into even more details in chapter 6.

⁵“Anonymous” functions are functions without a name = lambda abstractions.

2.1.4 Function type

Function values of type $\tau_1 \rightarrow \tau_2$ have the form $\lambda x^{\tau_1}.t$, where t is a term of type τ_2 , which may contain variable x of type τ_1 . When types are clear from the context, we will omit the type of the variable.

Given terms:

$$t_1 : \tau_1 \rightarrow \tau_2$$

$$t_2 : \tau_1$$

We have:

$$t_1 t_2 : \tau_2$$

Terms as described above represent function application. In the expression `lam F` in the table below, the variable `F` stands for a function in Lambda-Prolog notation again.

	Text notation	Representation in implementation
type	$\tau_1 \rightarrow \tau_2$	TP1 --> TP2
value	$\lambda x.t$	lam F
term	$t_1 t_2$	app T1 T2

2.1.5 Type of recursively defined functions

Recursive functions are typed similarly to non-recursive ones. Given term $t : \tau_2$, which may contain variables $x : \tau_1$, we can form (types may be omitted in the text):

$$\text{rec } f^{\tau_1 \rightarrow \tau_2}.(\lambda x^{\tau_1}.t) : \tau_1 \rightarrow \tau_2$$

Representation in the implementation:

`rec F`

Function application works the same as for non-recursive functions.

This time the function `F` in LambdaProlog representation takes two parameters as, for example, in `(f\X\ app f x)`. The reason is that we need a way to refer to the name of the recursive function in its function body — otherwise we would not be able to call it recursively.

It should be noted that there are other ways to specify recursive functions, as is done, for example, in [Han98]: there the recursive definition does not take a parameter, which means that we can create arbitrary recursive values (e.g. cyclic lists), too.

However, this generality comes with some problems in strict languages (not so in call-by-name ones). For example, it is not clear what the value

$$\text{rec } x.x$$

should mean. It could be of any type, but if we want to access it, it is not defined! This could lead to looping programs or, even worse, to crashing ones in real implementations, where the runtime system would just try to access an uninitialised memory location. Lazy languages do not suffer from the latter problem, because they evaluate all expressions lazily: the runtime system would not assume initialised locations and would either just keep looping trying to evaluate this example or (in some other cases) just delay evaluation of recursive subexpressions.

On the other hand, if we allow for a list constructor `::`, the following could be interpreted as the infinite list of ones:

$$\text{rec } \text{ones}.1 :: \text{ones}$$

Some strict languages (e.g. OCaml) allow such forms and impose restrictions on the right-hand side of definitions that prevent ill-formed cases. In practice it seems to be of hardly any use to allow such “infinite” data-

structures in strict languages⁶. It might indeed make some termination proofs more difficult, since inductively defined datatypes could then have “infinitely large” values, which does not fit well to their usual properties (they allow inductive proofs).

Therefore, we adopt the style that SML goes and disallow fully general recursive definitions completely. In other terms: evaluation always has to be triggered explicitly by a function application, thus preventing any problems with definiteness.

2.1.6 Recursive types

Recursive types are needed to specify inductively defined datastructures such as lists, binary trees, etc., or co-inductive ones like e.g. streams.

Here, for example, a way to encode lists in our language (text representation):

$$\begin{aligned} \mathit{nil} &\equiv \mathit{abs}(\mathit{inl}()) \\ \mathit{cons}(n, l) &\equiv \mathit{abs}(\mathit{inr}(n, l)) \end{aligned}$$

'abs' abstracts the type of the list representation and yields a recursive type. 'inl' just takes the unit-value as parameter, which indicates that we cannot get any more information from it (it stands for the empty list 'nil'). 'inr' takes a pair of values: the first one stands for the contents of a list element, the second one for the tail (rest) of the list⁷. If we want to implement a function on lists, we would have to use the keyword 'rep' on list values to “know” whether they represent empty lists 'inl()' or some cons-element 'inr(value, rest)'.

⁶As we mentioned earlier, however, one can use thunking techniques to simulate this in a strict language.

⁷Scheme and LISP programmers would call this a *cons-cell*.

It is important to note that using simple sum types together with pairs and the unit type, we have the potential to specify isomorphic representations for *all* kinds of recursive datatypes! This means that it is not necessary to come up with a more elaborate implementation of recursive types: it is not too difficult to translate any kind of existing recursive type with arbitrary data constructors into this representation and back again.

	Text notation	Representation in implementation
type	$\mu X.\tau$	mu TF
value	$\text{abs}(c)$	abs_rtp C
term	$\text{rep}(t)$	rep_rtp T

The variable TF stands for a function from types to types. As was the case with recursive functions, we need this to bind the recursive type within the body of its definition.

2.2 Abstract syntax

Having given an overview of the language, we can now present the full abstract syntax that will be used throughout the text (please refer to the informal specification in section 2.1 to learn about the representation used in the implementation).

$t ::= x^\tau$
$*$
(t_1, t_2)
$\text{fst}(t)$
$\text{snd}(t)$
$\lambda x^\tau. t$
$t_1 t_2$
$\text{inl}(t)$
$\text{inr}(t)$
$\text{case } t \text{ of } \text{inl}(x).t_1, \text{ inr}(y).t_2$
$\text{abs}(t)$
$\text{rep}(t)$
$\text{rec } f^{\tau_1 \rightarrow \tau_2}. (\lambda x^{\tau_1}. t)$

Table 2.1: Abstract syntax

2.3 Formal typing rules

The formal typing rules are to be read as follows: each rule consists of *premises* and a *conclusion*, the premises being written above and the conclusion below the solid line. To prove that a term is well-typed, we have to derive this using the rules until all derivations end in rules that do not require any premise to be true, these last rules being called *axioms*.

$\overline{x^\tau : \tau}$	$\overline{* : unit}$	
$\frac{t_1 : \tau_1 \quad t_2 : \tau_2}{(t_1, t_2) : \tau_1 * \tau_2}$	$\frac{t : \tau_1 * \tau_2}{fst(t) : \tau_1}$	$\frac{t : \tau_1 * \tau_2}{snd(t) : \tau_2}$
$\frac{t : \tau_2}{\lambda x^{\tau_1}.t : \tau_1 \rightarrow \tau_2}$	$\frac{t_1 : \tau_1 \rightarrow \tau_2 \quad t_2 : \tau_1}{t_1 t_2 : \tau_2}$	
$\frac{t_1 : \tau_1}{inl(t_1) : \tau_1 + \tau_2}$	$\frac{t_2 : \tau_2}{inr(t_2) : \tau_1 + \tau_2}$	
$\frac{t : \tau_1 + \tau_2 \quad t_1 : \tau \quad t_2 : \tau}{\text{case } t \text{ of } inl(x).t_1, inr(y).t_2 : \tau}$		
$\frac{t : \tau[\mu X.\tau/X]}{abs(t) : \mu X.\tau}$	$\frac{t : \mu X.\tau}{rep(t) : \tau[\mu X.\tau/X]}$	
$\frac{t : \tau_2}{rec f^{\tau_1 \rightarrow \tau_2}.(\lambda x^{\tau_1}.t) : \tau_1 \rightarrow \tau_2}$		

Table 2.2: Formal typing rules

Attentive eyes may have spotted the interesting pattern of the typing rules for function application and abstraction: they look the same as the logic

rules for implication introduction and elimination in natural deduction proof systems. This important relation, known as *Curry-Howard isomorphism*, relates proofs and beta-reduction (i.e. “evaluation”) of typed lambda terms, which turns out to be a very powerful tool in both computer science and logic.

2.4 Operational semantics

There are several ways to specify evaluation of programs, the operational semantics of a language. However, the *structural operational semantics* has the advantage of being *syntax-directed*. This ensures that it fills its purpose of providing for a strict guideline of implementation: it directly associates evaluation steps with each piece of abstract syntax in a formal way. Before this kinds of semantics became more widespread in use, it was common practice to specify the operational semantics by providing an abstract machine that interprets it. This is, of course, not so rigorous an approach from a formal point of view.

We will see in chapter 6 that our system implements both “normal” evaluation rules and partial evaluation as well. The declarative reading of the evaluation rules in LambdaProlog corresponds exactly to the rules we are about to specify: this makes it trivial to prove our implementation correct and demonstrates the very high level of programming achievable in LambdaProlog, making it most suitable for such tasks.

Before presenting the evaluation rules, we will give a short explanation of the meaning of *values*.

2.4.1 Values

Some terms that do not have free variables in them (i.e. they are *closed*) and if they are well-typed, correspond to basic elements of their types. In other

terms: they are values.

$c ::= *$
(c_1, c_2)
$\lambda x^\tau. t$ iff the whole lambda term is closed
$\text{inl}(c)$
$\text{inr}(c)$
$\text{abs}(c)$

Table 2.3: Values

$*$ is the only value of type *unit*. The pair (c_1, c_2) is a value when its type is $\tau_1 * \tau_2$ and when c_1 has type τ_1 and c_2 has type τ_2 . Any closed and well-typed lambda term $\lambda x^\tau. t$ is a value, as are $\text{inl}(c_1)$ and $\text{inr}(c_2)$ when their type is $\tau_1 + \tau_2$ and if c_1 has type τ_1 and c_2 has type τ_2 . A term $\text{abs}(c)$ is a value if c is a value of type $\tau[\mu X. \tau / X]$.

Terms which are values have the property that evaluating them does not change them: they are self-describing.

It is worth noting that in our call-by-value language variables always stand for values in the sense that their meaning is always already computed before one can access it: e.g. variables standing for function parameters are already computed before the function body in which they are bound is executed. This property is very important to know in program transformation: it guarantees that accessing variables will not cause any side effects, be it non-termination or others. We will come back to this in chapter 4.

2.4.2 Structural operational semantics

This subsection will present all the derivation rules to evaluate programs of the language. They are given in the form of *structural operational semantics*, which is sometimes called *natural semantics* of the language. Structural operational semantics rigorously defines each computational step to be performed after another so that an interpreter can be implemented from the specification in a straightforward manner.

Results derivable from the structural operational semantics include (the arrow \rightarrow is to be read as “derives to”):

Type consistency If $t : \tau$ is closed and $t \rightarrow c$ then $c : \tau$.

Determinacy For any closed t , there is *at most* one c such that $t \rightarrow c$.

Proofs can be obtained by induction on the structure of derivations.

Evaluation rules

As in the subsection on typing rules, the derivation rules of the operational semantics consist of premises and a conclusion. The elements of the rules specify evaluation steps. To evaluate a term on the left side of an evaluation arrow, we first have to derive (evaluate) all elements of the premise above it. Doing this may bind results of evaluation to variables, which may be used on the right-hand side of the conclusion. The right-hand side of the derivation is always a value!

It can happen that the derivation tree would be infinite in size, that we never meet a final rule during derivation: in this case we say that the term does not *converge* or that evaluation does not *terminate*. It is possible to prove equivalence of programs by structural induction or (in more difficult cases) by showing that they have equivalent derivation trees (induction on derivations). This can be quite tedious to do in this kind of semantics because

of the low level (computation steps) it addresses. *Denotational semantics*, which we will explain in section 2.5, is a more elegant way of establishing equivalences between programs.

$$\begin{array}{c}
\overline{* \rightarrow *} \\
\\
\frac{t_1 \rightarrow c_1 \quad t_2 \rightarrow c_2}{(t_1, t_2) \rightarrow (c_1, c_2)} \qquad \frac{t \rightarrow (c_1, c_2)}{\text{fst}(t) \rightarrow c_1} \qquad \frac{t \rightarrow (c_1, c_2)}{\text{snd}(t) \rightarrow c_2} \\
\\
\overline{\lambda x.t \rightarrow \lambda x.t} \qquad \frac{t_1 \rightarrow \lambda x.t \quad t_2 \rightarrow c_1 \quad t[c_1/x] \rightarrow c}{t_1 t_2 \rightarrow c} \\
\\
\frac{t \rightarrow c}{\text{inl}(t) \rightarrow \text{inl}(c)} \qquad \frac{t \rightarrow c}{\text{inr}(t) \rightarrow \text{inr}(c)} \\
\\
\frac{t \rightarrow \text{inl}(c_1) \quad t_1[c_1/x] \rightarrow c}{\text{case } t \text{ of } \text{inl}(x).t_1, \text{inr}(y).t_2 \rightarrow c} \\
\\
\frac{t \rightarrow \text{inr}(c_1) \quad t_2[c_1/y] \rightarrow c}{\text{case } t \text{ of } \text{inl}(x).t_1, \text{inr}(y).t_2 \rightarrow c} \\
\\
\frac{t \rightarrow c}{\text{abs}(t) \rightarrow \text{abs}(c)} \qquad \frac{t \rightarrow \text{abs}(c)}{\text{rep}(t) \rightarrow c} \\
\\
\overline{\text{rec } f.(\lambda x.t) \rightarrow \lambda x.(t[\text{rec } f.(\lambda x.t)/f])}
\end{array}$$

Table 2.4: Structural operational semantics

2.5 Denotational semantics

Denotational semantics provides for a framework in which program equivalences can be established using mathematical (equational) reasoning. The building blocks of this approach can be found in *domain theory*, which is concerned with complete partial orders (domains). In short, denotational semantics maps syntactic constructs of a language to least fixed points⁸ in domains, the elements of the domain representing different meanings. Since there can only be exactly one least fixed point (it is defined by its “leastness”), denotational semantics allows us to unambiguously assign meaning to computer programs.

A very important requirement for a denotational semantics is that it agree with the operational semantics on observations of interest. This means that if a term converges under the operational semantics (= evaluation terminates), it also does so under the denotational one and vice versa. Of course, the resulting value should be identical in both cases. Conversely, if evaluation does not terminate, this should be reflected in both semantics. Only in the case that the denotational and operational semantics fully agree can we draw sound conclusions about program equivalences. It should be noted that this does not mean that denotational semantics always allows us to establish (operational) program equivalences in the general case: though it is *sound*, it is not *complete* in this respect.

The existence of recursive types in our language requires us to make use of a special form of domains, namely so-called *Scott domains*, also known as *information systems*. The theory behind them is quite heavy, and an intro-

⁸It is not necessarily the case that a domain has a least fixed point, but the denotational interpretation of types usually *lifts* the domain of the basic types to include a least element (called *bottom*). This domain is then called a *complete pointed partial order*. The bottom element stands for the “unknown” value, for example, when a term does not converge.

ductory description alone would exceed the scope of this work⁹. It suffices for our purposes to point out the intention, which is to find least solutions in *recursive domain equations*.

It can be shown that information systems are indeed domains containing a least element. All the interesting types of our language, like product types, sum types, function types and, of course, recursive types can be brought into this framework, which allows us to specify an unambiguous denotational semantics for our language.

Readers who are interested in e.g. proving transformations correct are well-advised to make use of this powerful representation for the semantics of formal languages.

Because it would be necessary to explain a large part of the formalism behind the somewhat complex structure of *information systems*, we refrain from giving a full specification here. It should be pointed out another time that both the examples of this work and the implementation of the system follow exactly the formal specification of a strict language with recursive types of section 13.1 in [Win93], to which the interested reader may refer.

⁹For details, see chapters 12 and 13 in [Win93].

Chapter 3

Automated Transformations

This chapter will present an overview with examples covering the most common techniques applied in transformation of functional programs¹.

We will start out with *partial evaluation*², a basic technique for optimising programs, on which most other techniques can build. It exploits information that allows (partial) evaluation of programs before input is available. The implementation of our system mainly focuses on this technique, implementing it in a very general manner.

This will be followed by a presentation of a general strategy for transforming programs which are described by recursive equations: the *fold/unfold* method. Some of its special cases like deforestation (used for eliminating redundant intermediate datastructures) and the tupling transformation (can lead to nonlinear speedups by factoring out common computations in recursive calls) will be treated separately. Transformations of the fold/unfold type are of special interest to us: we will consider them in a later chapter on correctness and also give hints concerning their possible implementation in

¹Readers with knowledge of German might want to take a look at [Erw99], who not only provides for a good introduction to functional programming in general, but also gives an overview of the most common transformation techniques with many examples.

²See [JGS93] for a thorough introduction to partial evaluation.

the corresponding chapter.

Finally, we will consider the *Bird/Meertens Formalism*, which exploits the fact that many functional programs are constructed out of specific building blocks. Mathematical properties of these building blocks allow us to rewrite their combinations to simpler forms, thus eliminating unnecessary computation.

3.1 Partial evaluation

3.1.1 Example

It is probably best to explain this method by example:

```
λx.  
  (rec f.λy.  
    y (case x of  
        inl(l).inl(l)  
        inr(r).inr(snd((f,r))))))  
  (case (inl(λx.x)) of  
    inl(l).l  
    inr(r).fst(r))
```

It is fairly difficult to figure out what this function (lambda abstraction) does by just taking a short look at it: it uses higher-order functions, recursive functions, case statements and manipulation of pairs to compute its result.

In fact, the meaning of this program is relatively simple, i.e. can be computed by another function that does not need this many computation steps:

```
λx.x
```

It is just the identity function! How can we derive this result? First of all, it is interesting to note that the program is a value: it is a lambda

abstraction. Applying the usual evaluation rules³ to it will not change the result. This is not the way in which we can improve it.

Therefore, to be able to optimise, we need rules that are able to “step” into the body of the function⁴ and enable us to find out, which of its parts could be evaluated statically: another term used for this is *binding-time analysis*. Having identified the parts that can be evaluated before the program gets its full input, we can apply simplification rules to those statically computable parts and specialise the program.

Indeed, if we take a second look at the program, we see that the second case statement can be evaluated immediately: the value of the argument is `inl($\lambda x.x$)`. Therefore, considering the first case arm, the meaning of the whole second case statement is `$\lambda x.x$` .

This function is passed to the recursive function (as a higher-order function) and bound to `y` in its body. There it is applied to the meaning of another case statement. This time it is a bit more tricky to find out, what this case statement means, because we do not know `x`, the value of sum type on which we perform the case switch, at compile time!

The first case arm can obviously not be partially evaluated, but the second can: the pair inside is known at compile time, and we see that the second element is requested. Therefore, the second case arm can be evaluated to `inr(r)`. This is an interesting point. The partially evaluated case statement looks now as follows:

```
case x of
  inl(l) . inl(l)
  inr(r) . inr(r)
```

³See page 27 for the evaluation rules (structural operational semantics).

⁴As we will see in chapter 6, the language we use to implement transformations (LambdaProlog) has very convenient builtin functionality to support descending into bindings.

Whatever x is ($\text{inl}(l)$ or $\text{inr}(r)$), the same will be returned again. Thus, we can simplify this case statement to just x alone without even considering its concrete value. Applying the higher-order function to it that we mentioned earlier (it is the identity function), we get x again — and have fully derived the identity function for the whole function as required. Each step in this derivation was actually fairly easy to follow, though it is somewhat tedious to do this manually.

3.1.2 Properties of partial evaluation

Partial evaluation usually does not lead to dramatic speedups due to various reasons: first of all, hardly any programmer would seriously write code as in the example above. Furthermore, none of the partial evaluation steps presented above eliminates more than a constant overhead. This means that speedups are typically of linear nature.

Still, this method is very important for transformation systems in general: many advanced transformations result in programs which can be improved by partial evaluation, e.g. the instantiation rule that is used by the *fold/unfold* strategy⁵.

Since advanced transformations usually involve some kind of search process, interleaving some of their applications with partial evaluation may significantly reduce the size of the search space. This was the main intention behind implementing a general purpose partial evaluator in this project.

A point which we have not touched in the whole discussion of partial evaluation so far is *correctness*. The reader may wonder, why this is an issue, as there did not seem to be any dangerous transformations involved in our example above: all of them clearly preserved the meaning of the program.

Unfortunately, the semantic properties of our language, it is a strict one, can lead to situations where this does not hold in general. Additionally,

⁵See section 3.2.

most real implementations of functional languages allow side effects like I/O or exceptions, which makes this issue even more complicated. Because of the importance of correctness of transformations, we will need to discuss this topic in more detail by examples in chapter 4.

A computational formalisation of partial evaluation as given in [HD97a] and [Hat98] seems to provide for a very strong basis to improve partial evaluation techniques in the presence of problems as mentioned above. Our implementation⁶ takes up this elegant approach and shows that this yields a most declarative view of partial evaluation in our logic implementation language LambdaProlog.

3.2 Fold/Unfold method

In their seminal paper [BD77], Burstall and Darlington developed a general strategy, often referred to as *fold/unfold method*, for transforming functional programs that are represented as *recursive equations*. Many researches subsequently took up this approach and refined it in several variants.

Due to space restrictions, we will not give any fully worked example transformations⁷ of this method or of any of its variants. In later chapters we should, however, consider specific patterns that can arise during transformations, where we show problems concerning correctness or give hints how to implement them using the developed transformation framework.

Here an example of recursive equations that specify how to calculate the sum of elements in a list ('[]' stands for the empty list, ':' is the list constructor):

⁶See chapter 6.

⁷A very complete discussion of fold/unfold transformations and their variants with many and impressive examples can be found in [PP96]. [JGS93], too, has a fairly detailed chapter on it.

$$\text{sum } [] = 0$$

$$\text{sum } (h::t) = h + \text{sum } t$$

The fold/unfold strategy consists of a set of rules that, when applied wisely, allows a very large class of optimising transformations on such recursive equations. As the word “wisely” indicates, however, the degree of automation is not always as high as one might hope: some examples require the discovery of so-called *Eureka*⁸-steps, steps which cannot be derived using the rule set, but rather require insight into the problem that may not be easily automated.

Here is a short overview of the rules (see [PP96] for their detailed properties) - it also mentions the correspondence of the rules between recursive equations and our language representation.

Definition Rule

This rule allows adding new recursive equations to a program. In our language this would correspond to adding new functions. This is necessary, because some optimisations might require treatment of a computation in a separate function.

Unfolding Rule

Unfolding an expression means replacing it with the right-hand side of an equation whose left-hand side matches this expression. Variables in the replaced expression are substituted for values which were captured during matching. In our representation of the language this would correspond to function application: substituting the formal parameter of a function for its argument in the function body. This results in a new equation (a new function term) in which the expression is replaced.

⁸“Eureka!” was the word that the Greek philosopher Archimedes supposedly cried out when he discovered the relation between the important physical concepts of weight and density.

The important aspect to be careful about is that unfolding is only sound with respect to our strict semantics when evaluating the parameters of the function does not yield any side effects like non-termination, I/O, etc.⁹ Not respecting this may, for example, transform non-terminating programs to terminating ones.

Folding Rule

This is the opposite of the unfolding rule: we match an expression against the right-hand side of an equation and replace it with the left-hand side of the same, again substituting values that were bound in the matching process for the variables in the replaced expression. This means with respect to our language: we replace an expression with a call to a function whose body (right-hand side of a recursive equation) matches this same expression. This should in most cases preserve the meaning of the equation (the function term) in which the expression was replaced. Here we have a dual problem to the one of the unfolding rule: as we will see in our chapter on correctness, folding may transform terminating programs to non-terminating ones in certain cases.

Instantiation Rule

Application of the instantiation rule introduces an instance of an already existing equation. A small example: taking our sum-example from above and given the following rather redundant definition:

$$\text{sum}'\ l = \text{sum}\ l$$

we can apply the instantiation rule to `sum'`, because we know all the instances of `l`, which is of type `list`. We can instantiate twice, once using `'[]'` and a second time using the cons-operation `': :'`.

⁹We will learn more about this in chapter 4.

```

sum' [ ] = sum [ ]
sum' (h::t) = sum (h::t)

```

It should be noted that the right-hand sides of the new equations have more “information” now: they “know” what kind of argument is passed to the sum function, which could be exploited using a recursive unfolding step and partial evaluation followed by a folding step to simplify the function in such a way that the redundant function call to 'sum' is completely eliminated. The resulting function would be equivalent to 'sum' itself.

The instantiation rule essentially works on sum types¹⁰. Instantiating other types of values (e.g. pairs) alone does not offer information to the function body. The only way that choice can enter the system (information corresponds to choice) is through case statements that decide, which of two choices to take (depending on a sum type). Of course, pairs may also contain elements that have sum type. In this case, the instantiation rule may try all possible ways to instantiate the pair that contains them (recursively as required with nested types). Here an example, how the instantiation rule¹¹ would work in our language. We assume that f is a function of type $(\tau_1 + \tau_2) \rightarrow \tau_3$ and x is of type $\tau_1 + \tau_2$:

$$\lambda x. \lambda f. f \ x$$

We specialise the argument to f within two case arms:

¹⁰See 2.1.3 again for their definition.

¹¹We will call the instantiation rule *specialisation rule* in our language so as to prevent confusion with instantiating existentially quantified variables when we do transformations on programs in LambdaProlog.

```

λx.λf.
  case x of
    inl(l).f (inl(l))
    inr(r).f (inr(r))

```

This might allow a partial evaluator to proceed by applying the function f to the specialised value.

Where-Abstraction Rule

The where-abstraction rule introduces definitions which are local to a given recursive equation. It is actually not so much different from the *definition rule* for our purposes, because our language does not have any extra constructs to support this rule, anyway: lambda abstraction and function application can be used to achieve the same effect of binding definitions. However, we can (and will) extend the language with a so-called *let*-construct, which can come handy in some occasions as we will see in chapter 4. We leave explanation of further details to [PP96].

Algebraic Replacement Rule

This rule allows exploiting equivalences of expressions to rewrite them into each other. For example, if we know that the program handles natural numbers and if we have defined a multiplication function for it, we can make use of the mathematical property of it being commutative: we could swap around parameters to the multiplication function, which might allow other transformation rules to match.

It should be noted that this rule is potentially very computationally intensive, but applying it in full generality it could allow the most powerful transformations, exploiting all kinds of equivalences.

3.2.1 Deforestation

A specific instance of the *fold/unfold*-strategy, namely deforestation, was developed by Phil Wadler in his often cited paper [Wad90]. It addresses a problem associated with high level programming in functional languages: computations are “glued” together via intermediate datastructures. Yet, these intermediate datastructures do not necessarily represent parts of the result itself. This can be best shown with an example:

```
sum (map sqrt l)
```

'map' takes a function (here: 'sqrt', the square root function) and applies it to every element in list 'l', which results in a new list that contains all these square roots. 'sum' takes a list and sums up all elements. This is a very convenient way of writing code, much clearer than this version:

```
sum_sqrts [ ] = 0
sum_sqrts (h::t) = sqrt h + sum_sqrts t
```

However, the last version is more efficient instead: whereas 'map' produces (allocates) an intermediate list just to remove (deallocate) it immediately after the computation, 'sum_sqrts' does not. Allocating and deallocating memory is usually a very time consuming task and may also raise the memory requirements of the program. Therefore, we would like to have transformations that eliminate those intermediate lists (and possibly other kinds of intermediate datastructures, e.g. trees).

Wadler gives a set of rewrite rules for the deforestation algorithm and proves that transformations always terminate and that the resulting program is guaranteed to be at least as fast as the input program (= no performance loss). Still, for the deforestation algorithm to work, the program and its terms must fulfill certain requirements: the functions out of which a term is

constructed must be in a special (“treeless”) form¹², which also requires that they be linear¹³.

In the general case, Wadler’s method does not scale up to higher-order functions, which is a significant shortcoming in languages whose expressiveness heavily depends on them. Even though allocating datastructures can be computationally expensive, this can be done with only a constant factor of overhead. Thus, the performance gains of deforestation are typically only linear in nature.

Several related techniques have come up over time, which try to improve the method in different aspects: e.g. [LS95], who explains the advantage of focusing on so-called *catamorphisms*¹⁴, a special pattern of recursion, which does not require so much search when looking for a point to apply the folding-rule.

3.2.2 Tupling transformation

The tupling transformation is another specialised technique derived from the fold/unfold-strategy. It allows introduction of accumulators into functional programs, which can have a tremendous impact on performance. Here a typical example¹⁵, where it leads to such improvements:

¹²The somewhat lengthy exact definition of this term can be found in [Wad90].

¹³A linear function does not use its parameter more than once, considering different case arms separately. We will see in chapter 6 that our partial evaluator can indeed simplify applications of nonlinear functions without causing performance penalties and does not even have to check the functions for linearity to do this: our approach seems to cope very well with such cases, too.

¹⁴A good introduction to the various uses of such and related constructs is given in [MH95].

¹⁵These examples are given in the pure and lazy functional programming language *Haskell*, because it resembles the definition of recursive equations most. This allows readers to try them out. For more information on Haskell, see the URL: <http://www.haskell.org>

```

fib n =
  if n <= 1 then 1
  else fib (n-1) + fib (n-2)

```

As the name indicates, this function computes the Fibonacci numbers. As declarative as this definition is, as inefficient is its execution as a program. The time complexity of this program is exponential due to the two recursive calls that are done in its body. But taking a closer look at this definition, we see that a significant part of this computation is redundant: 'fib (n-1)' requires the computation 'fib (n-2)' already so there is no need to recompute the value of the latter in the second call.

Introducing a tuple which accumulates the last two values (more is not needed), we can save the call to 'fib (n-2)'. This effectively eliminates this call from the program: instead of branching into two calls in each of the recursive steps, which causes the exponential time behaviour, there will be only one, which makes the function run in linear time. This is a significant improvement over the first algorithm:

```

fib2 n =
  if n <= 1 then 1 else aux (1,2) 2
  where aux (n2,n1) m = if n == m then n1
                        else aux (n1, n2+n1) (m+1)

```

As was mentioned, there is only one recursive call left. But we also see that this algorithm is much more complicated to understand than the first one. This demonstrates that more efficient algorithms can indeed be difficult to derive.

The general application pattern of the tupling strategy starts out trying to discover computations which have to be computed repeatedly in different recursive calls or at different levels of recursion. It factors those out and remembers them in an accumulator which is passed from each recursive call to the next one. Thus, the expression has to be computed only once, which

can lead to considerable performance improvements, especially if this means that one or more recursive calls can be eliminated.

People who think that the efficient implementation of the Fibonacci function was easy to derive for them and is optimal might consider the example in the appendix on page 103. It is truly a monster of a function: if it was not for the name, hardly any programmer would have the faintest clue of what it does without running it. In fact, this function computes the Fibonacci numbers again exponentially faster than the one we have just derived! — It runs in logarithmic time! We will not go into the details of its derivation¹⁶, but this example should clearly demonstrate the very promising perspectives of functional program transformation: they even allow super-exponential speedups. Of course, much more research is necessary to make this power applicable to real world programs.

3.3 Bird/Meertens formalism

In contrast to the fold/unfold-strategy, the Bird/Meertens formalism¹⁷ has a much higher degree of automation: instead of having to discover important properties in Eureka-steps, this formalism exploits known mathematical equalities that hold for often used constructs, in particular for common higher-order functions¹⁸.

A simple example of this kind would be applying the list operation 'map' twice with two different functions ('o' stands for function composition):

$$\text{map } g \circ \text{map } f$$

¹⁶The impressed reader may learn this trick, which could indeed be automated using tupling techniques, in [PP96].

¹⁷A broad overview with many examples and a very good bibliography can be found in [Erw99].

¹⁸E.g. list operations like 'map', 'fold_left', 'fold_right', 'scan_left', etc.

This can be optimised to:

$$\text{map } (g \circ f)$$

which effectively eliminates an intermediate list. Many more such equations exist, and additional ones can often be defined for specific applications. Using tools for automatic verification or even generation of such rules would allow the system to become more powerful over time while still retaining a high degree of automation.

One minor drawback of this method without any further extensions is that its effectiveness depends on the size of its rule set. This raises questions about how to control search: many rules require a significant effort in pattern matching. To be applicable to real world programs, clever heuristics for rule selection would have to be developed to overcome this problem. Another shortcoming is that it is not so generic by nature: the knowledge about transformations lies within the rule set. If the programmer does not make use of the known rule patterns, this method is unlikely to lead to significant improvements, whereas some members of the fold/unfold-family are often applicable to programs which involve datastructures that have never been seen before. Current research in this field is improving this situation, and there are indeed relations to offsprings of deforestation (e.g. fusion techniques as in [LS95]), where higher-order functions related to list operations like 'fold' are generalised to arbitrary datastructures automatically: this makes known equalities available in many general cases.

Chapter 4

Correctness of Transformations

Imagine the following scenario: a customer orders a mission-critical and efficiency-dependent system from your software company. You develop and test it with your compiler that makes use of advanced transformations to optimise it: everything works fine. Your customer receives your (untransformed¹) sources, compiles the program with a compiler that is proven to be conformant to the specification of the language, launches the mission — and suddenly, a multi-million dollar satellite crashes due to non-terminating² execution of a critical software component: your system.

How can it happen that termination behaviour changes if our transformations only do what the evaluation rules of the language tell them? Here is an example:

¹It should be pointed out that transformed sources are only in rare cases readable to humans. Therefore, if we want to be able to maintain them, we will have to rely on hand-written code or specifications. This disallows that we ship transformed source programs for which we do not have a maintainable version that is absolutely equivalent in its behaviour. Otherwise, maintainance would be close to impossible.

²One might be tempted to accept the opposite: that a program that “improves” on termination behaviour by some transformation (see the example on this page) is “better”. As we will see in section 4.2, this does not address the more general problem of arbitrary side effects and might lead to difficulties with software maintainance.

`fst (*, f x)`

At first sight it may seem that we can simplify this expression to `*` only, as the semantics of `fst` and pairs would indicate. However, what about the application of the unknown function `f` to the unknown parameter `x`? What happens if it does not terminate? Due to our strict semantics, which will evaluate all components of the pair before applying `fst`, evaluation of the whole expression would not terminate. But if we simplify, we change this behaviour: the program will terminate then!

As we have mentioned earlier, strict semantics makes it more difficult to apply transformations of this kind without changing the meaning of programs, and computational effects like non-termination are definitely part of it.

4.1 Preserving termination behaviour

First of all, it should be noted that termination issues are among the trickiest problems in computer science: it can be shown that there is no algorithm that can completely and consistently decide for all programs whether they terminate or not.

Because we can only be sure about *some* cases, whether evaluation of a term terminates or not, we have to assume the worst-case if we are not sure: this implies that there will never be a transformation system that can exploit all possibilities to simplify a program. In uncertain cases the system would have to refrain from applying a transformation, because it cannot prove that this will not change the meaning of the program — even if this were perfectly safe.

However, as we will see, not all is lost: we can change the way in which we improve the code by transforming it into other representations that give

more opportunities of simplification without destroying the effect behaviour of the program.

In the following subsections we will present more examples of problems associated with the correctness of various transformations and work out ways to get around them.

4.1.1 Correctness of partial evaluation

In our initial example we have already shown a case where naive application of partial evaluation does not yield the intended result. What can we do to improve the obvious inefficiency? As was mentioned in previous chapters, call-by-value evaluation does not allow transformations as straightforward as call-by-name, but it guarantees another property that we can exploit: variables always stand for reduced terms — that is what makes a language eager (strict). This means that in this example

$$\mathbf{fst} (*, v)$$

we can be absolutely sure that 'v' is bound to a value. This makes it perfectly safe to apply the obvious simplification. The only thing we need to make sure is that the term that possibly causes non-termination is still evaluated just for the purpose of preserving termination behaviour.

There are two ways in which we can achieve this. The first one does not require any change to the language:

$$(\lambda v. \mathbf{fst} (*, v)) (f\ x)$$

We just put the whole term that may contain a possibly non-terminating computation into a lambda abstraction, *lift* the computation out of the potentially diverging term and introduce it as a parameter of an application to this function. Because of strict evaluation, this parameter is guaranteed to

be evaluated before the body of the function is consulted. Now we can make use of our simplification for terms and arrive at:

$$(\lambda v. *) (f x)$$

This prevents us from changing the meaning of the program during transformation. However, we were required to add an unnamed function (a lambda abstraction). If we want to eliminate the penalty of having to evaluate the function term (lambda abstraction) before applying it³, the second choice we have is to introduce a new construct to the language: something similar to function application, but which does not need to evaluate function terms. Indeed, most functional languages feature a *let*-construct that allows the programmer to evaluate a term and substitute it within an expression. The last example would then look as follows:

$$\text{let } v = f x \text{ in } *$$

We will describe this and related constructs that we use to lift various computations out of terms in more detail in chapter 6. There we will also learn about formal aspects of such transformations.

4.1.2 Correctness of Fold/Unfold transformations

Unfolding

In the previous chapter we have described the various rules of the fold/unfold strategy and mentioned that two of them (the folding rule and the unfolding rule) may cause problems in certain cases. If we take a closer look at unfolding, this rule is actually just a special case of partial evaluation: if we

³As demanded by the operational semantics: there could be a more complicated term in this place.

know the definition of a function (its body)⁴ and given its argument, we can apply the function — but only in the case when evaluation of the parameter terminates. For example:

```
(λx. *) ((rec f.x. f x) *)
```

We see that the parameter to the right is constructed by a call to a recursive function that can never terminate. Therefore, it is not safe to apply the lambda abstraction to the left of this term, because the lambda abstraction does not use its parameter: it always returns the unit-value. This would again remove the effect of non-termination, which is not the intention of our transformation system. Of course, we can again use the trick mentioned in the previous section on partial evaluation and lift out potential non-termination, which might allow more simplifications. In the example above this would result in:

```
let v = (rec f.x. f x) * in *
```

The following example shows a more interesting case (we assume that '*suspicious_fun*' cannot be proven to terminate):

```
(λx.
  case x of
    inl(l) . inl(x)
    inr(r) . inr(*))
(inl (suspicious_fun *))
```

This could be transformed to:

⁴The only case where we cannot know the function body occurs when we use higher-order functions — when a function is taken as argument by another function. If the function passed as argument cannot be known until runtime when all data is available, we have no information about its body.

```

let v = suspicious_fun * in
(λx.
  case x of
    inl(l).inl(x)
    inr(r).inr(*))
(inl v)

```

and then simplified (e.g. using our partial evaluator) to

```

let v = suspicious_fun * in inl(v)

```

which is a significant improvement. For completeness it should be mentioned that we can (in some cases) *inline* such lifted computations again⁵. Here is the completely simplified result:

```

inl(suspicious_fun *)

```

Folding

Folding, calling a function that evaluates an expression instead of evaluating the expression directly, generally works without problems. There is, however, a specific and important case where this does not hold:

In many variants of the fold/unfold strategy the folding step is used to “tie a recursive knot”. This means that the folding rule is used to introduce a recursive call to the function containing the expression to be replaced! The rationale behind this step is that we sometimes want to “invent” a new recursive function that computes the same meaning as an initial one, but, for example, eliminates intermediate datastructures⁶.

When applied in a naive way, this can transform terminating programs to non-terminating ones. In the following example we assume that we want to eliminate intermediate datastructures in some function *f* by transforming it

⁵This will be explained in chapter 6.

⁶See section 3.2.1 on deforestation techniques.

to `'g'`. In an intermediate step we end up with the following definition, which treats each case of its input parameter (a recursive sum type⁷) explicitly:

```
rec g.x.  
  case x of  
    inl(l). f (inl(l))  
    inr(r). f (inr(r))
```

As should be obvious, this function `'g'` is semantically equivalent to `'f'`: it just already takes one choice (a case statement) before passing on the parameter to `'f'`. One could argue now: if it is semantically equivalent, why not call this function recursively now instead of relying on the initial definition of `'f'`? This step in which we “tie the recursive knot” would look as follows (the function calls to `'f'` are replaced with calls to `'g'`):

```
rec g.x.  
  case x of  
    inl(l). g (inl(l))  
    inr(r). g (inr(r))
```

It should be clear that we have rather tied a recursive gallows rope rather than a recursive knot: the function does not terminate for any input! Where is the hidden catch in our assumptions?

During this work it turned out that certain preconditions must be met before we can safely undertake a recursive folding step: the parameter to the function call must be free from explicit values of sum type (values constructed with `'inl'` and `'inr'` in our language), including nested datastructures that contain such values. Only variables of sum type are acceptable. This is not the case in the upper example: the argument to `'g'` is `'inl(l)'` and `'inr(r)'` respectively.

⁷We leave away the syntactic elements `'abs'` and `'rep'` (see section 2.1.6), which are only necessary for type checking recursive types and would otherwise just make the example unnecessarily verbose.

If this requirement is not fulfilled, this can indicate that the “original” function definition, which was called before the transformation, still handles information that the new definition cannot yet handle recursively⁸.

It may be, however, that it is still safe to try folding even if there are explicit values of sum type in the parameter term. This requires us to extend the criterion above to the following: the recursive function application that we obtain by “tying the recursive knot” and which contains explicit values of sum type must not be able to “reach itself” when called. This means that in the next cycle of the recursion, there must not be any evaluation path that could demand evaluation of the recursive knot again. If this is violated, a function parameter that happens to match this explicit value of sum type cannot be handled any more. Evaluation could stay in a loop that ranges from the outermost expression of the function body to the point where the folding rule was wrongly applied.

We could use the partial evaluator to find out easily whether this is the case: we “tie the recursive knot”, unfold this recursive call (= apply it) and let the partial evaluator try to simplify the resulting expression. If this yields another expression that *still* contains the “recursive knot”, the expression that was introduced by folding, then the application of the folding rule was not guaranteed to be invalid: loops may occur.

Since the power of the partial evaluator only depends on the ability of our system to infer non-termination (or other side effects — see the next section), we have limited the question of when folding is appropriate to this generally undecidable problem. This means that there will always be safe cases which we will have to reject for the sake of consistently preserving termination behaviour (and other effects).

Unfortunately, we do not (yet) have a formal proof of all the claims above,

⁸As we have already mentioned in earlier chapters, values of sum type encode information: they offer choice.

but the verbose explanation seems reasonable, and future work could try to address this question.

One detail should be noted: it may well happen that loops occur after folding even when the function parameter to the recursive call did not contain an explicit value of sum type. This loop, however, would then necessarily also happen without the transformation, which means that the initial program was not completely defined for all input. In other terms: folding is justified and might optimise a few cases for which the program is defined, and it would not change termination behaviour in any way.

Summing up the correctness issues concerning the folding rule:

- It is always safe to recursively fold functions whose arguments do not contain explicit values of sum type.
- If there is such an explicit value in the function parameter, one can use partial evaluation after unfolding the “recursive knot” once. If the resulting partially evaluated expression contains the recursive knot again, folding is not guaranteed to be safe (undecidable question).

4.2 Other side effects

As we have mentioned in the introduction to this chapter, transformations that “improve” termination behaviour are not acceptable: first of all, it is difficult to maintain a program when the transformed (and possibly not “human friendly”) code does not behave the same way as the untransformed version does: for example, it does not terminate on the same input values on which the transformed code terminates with a wrong result. Secondly (and more important): non-termination is not the only kind of side effect. Most strict functional languages contain “impurities” like I/O and exceptions (side effects that lead to observable behaviour). If we do transformations, it would

be fatal if one of those effects suddenly vanished, were duplicated or if their order changed, thus changing the meaning of the program. For example, let us assume that our language had strings and an I/O-function for printing them, this function returning the unit value after execution. We consider this example:

```
fst (*, print "Hello World!")
```

Although 'print' always terminates, we cannot just simplify this expression according to the semantics of 'fst': if we did so, the print statement would never be executed, which is clearly against the meaning of the program. Therefore, if we take a more general solution to the problem of effects, we must completely disallow transformations that manipulate the order in which they occur and how often they do. The approach mentioned earlier, namely lifting computations, works here equally well: this solution seems to be the most general way to correctly handle effects of all kinds.

This is also a good time to mention that the semantics of the language we have specified does not enforce any evaluation order: it does not have to, because it is *pure*: there are e.g. no I/O-statements. Order of evaluation is completely irrelevant in purely functional languages, which is the reason why they make equational reasoning so easy⁹.

For example, in the following piece of code it obviously makes a big difference whether the first element of the pair is evaluated first or the second:

```
(print "Hello ", print "World!")
```

If we wanted to specify evaluation order in the operational semantics, we would have to make this explicit by passing around a “state of computation”.

⁹Most strict functional languages are indeed impure (e.g. ML, Scheme, etc.). However, it is very rare that lazy languages are impure: the reason is that it is extremely difficult to predict when (or even whether) some expression will be evaluated. Therefore, a “lazy” and “impure” language would behave in rather nasty ways.

For the specific case of pairs, this might look as follows¹⁰:

$$\frac{\langle t_1, \sigma \rangle \rightarrow \langle c_1, \sigma' \rangle \quad \langle t_2, \sigma' \rangle \rightarrow \langle c_2, \sigma'' \rangle}{\langle (t_1, t_2), \sigma \rangle \rightarrow \langle (c_1, c_2), \sigma'' \rangle}$$

This would specify a left-to-right evaluation order: the state σ , which is associated to each expression within angle brackets, gets passed from evaluation step to evaluation step, possibly changing as it does so. If somebody wanted to extend our language with impure features, maybe so as to apply the implementation of our partial evaluator to more “realistic” languages, they should bear this (importance of evaluation order) in mind. The formal semantics of the language would need to be changed to capture the notions of state and evaluation order.

4.3 Practical correctness issues

So far it might seem that there are only theoretical questions concerning the validity of transformations. Unfortunately, a very important aspect seems to be lacking in the literature: the impact of transformations on the executability of programs on “modern” computers. This is a point which, as we will see, may have consequences as drastic as transformations that do not preserve the “mathematical” meaning of programs, including the formalisation of effect behaviour.

So as not to confuse the reader with too complicated a program in our spartan language, the following example is written in OCaml¹¹, an ML-dialect (ML is strict):

¹⁰[Win93] also includes a large chapter on the semantics of imperative languages (languages where execution of side effects is the central paradigm).

¹¹OCaml is currently available in the WWW at the URL: <http://caml.inria.fr>

```

let rec length1_aux accu list =
  match list with
  [] -> accu
  | head::tail -> length1_aux (accu + 1) tail
let length1 list = length1_aux 0 list

let rec length2 list =
  match list with
  [] -> 0
  | head::tail -> 1 + length2 tail

```

This example does not stand for a real case, it is somewhat simplified: we assume that 'length2' is the result of an optimising transformation from 'length1'. It does in fact not differ in efficiency from a theoretic point of view. In reality problems as we will describe further below might involve more than one function, for example in a fusion transformation.

Function 'length1' makes use of an auxiliary function which computes the number of elements in a list in a tail-recursive way. This allows good compilers to transform the recursive function call to a loop instead: this fits much better to the execution model of von Neumann architectures. If it cannot do so, it would have to throw the current state of computation for every recursive call on a stack whose space is bounded (indeed, normally bounded quite tightly).

Unfortunately, if this stack space is exhausted, the program will crash: a very severe form of failure. This means that a transformation that takes one or several tail-recursive functions, optimises them and produces an efficient version that is not tail-recursive any more (e.g. 'length2' in the above example) changes the behaviour of the program as observable on a real machine drastically: running the example with a test list of length 50,000 on a current machine using the current compiler allows both version to terminate

correctly. Doubling the size of the list, however, crashes 'length2' only, while the original version terminates correctly. A list of 100,000 elements can be fairly little for a real-world task. The reader having new transformations in mind should be careful to avoid ones that lead to such behaviour.

Chapter 5

Controlling Transformations

Not only do we want to obtain efficient programs by transformation: for practical purposes it is necessary to guide the application of transformations in such a way that the transformation process terminates in acceptable time.

First of all, it is important to note that this has nothing to do with correctness of transformations: either the preconditions for a correct transformation are met and it could be applied — or not. Control of transformations is necessary when the preconditions for *several* of them can become true at the same time, in which case we say that we have *choice* between transformations.

Unfortunately, this freedom of choice is rather a burden here: we not only may choose, we must choose! The problem is that once we have decided for a specific transformation and apply it, we may lose the opportunity to apply any of the others — and might even get further away from the goal, an optimal transformation, if our choice was bad. In such cases we would have to take back some or even all of our choices and try others, which means that the transformation effort so far was wasted.

This situation is especially bad if we consider that after each transformation we again have several to choose from. This clearly implies that the number of choices available to us can rise exponentially, which can easily make the transformation process intractable for anything but toy problems.

5.1 Reducing the number of choice points

The first kind of technique we can try to make search more tractable is to eliminate choices that are sure to lead us away from the goal. For our purposes of program transformation it is noteworthy that there are potentially infinitely many programs that have the same meaning. It would be helpful to only have transformation rules that handle some of these cases rather than all and to use specific rules which do not require search that can transform any of the many equivalent variants into ones that can be handled. This might allow us to reduce the number of transformation rules we need.

5.1.1 Partial evaluation

One very useful approach is using a partial evaluator to eliminate as many statically computable computations as possible. Clearly, this redundancy should be eliminated in any case, and partial evaluation transforms many equivalent programs into a common representation, thus also removing the need for having to cope with all of them in other transformations. Because many rules can often be applied to different parts of a program, the partial evaluator is likely to reduce the number of some of these cases.

But does partial evaluation not require any search? Although it requires pattern matching in our implementation, which, of course, needs computation time, once a rule of the partial evaluator matches, it is guaranteed to be the only rule that could match in this place¹. This means that there is only one possible choice, which implies that there is no true search involved.

One might raise the argument that there is choice, for example, when partially evaluating pairs: we could choose one component only. This, however, is dangerous: the partial evaluator is supposed to preserve the effect behaviour, the latter depending on evaluation order. Therefore, we must

¹We will describe the rules of the partial evaluator in more detail in chapter 6 and 7.

never abuse the partial evaluator to work on subterms in such a way that effect behaviour may be changed. This means that we will always have one node in the abstract syntax tree for which we know that it is safe to apply partial evaluation to it (in the worst case it is the root node). Taking one node beneath it would be unsound — effects might be lost. Taking one above it does not improve the performance of the partial evaluator, rather degrade it: it would have to partially evaluate a larger tree unnecessarily. Therefore, we always only have one rational choice.

One could imagine that there are other transformations that have this property of not introducing alternatives when applied. Because such transformations do not induce nonlinear time behaviour due to the lack of search whereas others do, it is very cheap to apply them: they will only impose a constant factor on the execution time of the transformation system.

Thus, it is advisable to run the partial evaluator every time when statically computable parts could be present. This is definitely a good idea when the initial program is given, but also after each other transformation that might reveal new information to partial evaluation. For example, the partial evaluator never applies recursive functions whose termination it cannot prove². Other transformations, e.g. unfolding of recursive functions, could allow the partial evaluator to proceed again and optimise the newly transformed part.

Partial evaluation has another fine property: it uncovers useful information that is hidden in complicated constructs. For example, it can find out which parts of the program do not necessarily terminate or yield side effects³. This information could again be used to remove superfluous transformation alternatives, for example, by eliminating duplicated computations (common

²We have noted elsewhere that the unfolding rule is a special case of partial evaluation. However, so as not to cause the partial evaluator to loop when handling recursive functions, we must handle this case of unfolding recursive functions separately.

³See chapter 6 for details on how the partial evaluator achieves this.

sub-expression elimination), which is also implemented in our system.

5.1.2 An advanced technique for guiding search — rippling

Viewing transformations as rewrite rules in general, an interesting method for guiding search is *rippling*⁴. Its purpose is to maintain a specific pattern in a term during transformations while eliminating other subterms or at least moving them to designated positions where they do not disturb further transformations.

The rationale behind this technique is that it sometimes happens that the application of a promising rule is obstructed by terms which disallow the pattern of the rule to match. By protecting the pattern using special annotations and by only allowing application of rules⁵ that carry similar annotations which have to match, too, it is guaranteed that only rules that do not destroy the target pattern can be applied.

Due to correctness requirements imposed on the annotations that guarantee a strictly decreasing measure on a well-founded order during transformation steps, the transformation process either succeeds in the target configuration or it ends in a state where no rule is applicable any more. This means that the transformation process is guaranteed to terminate at some point of time. Because the annotations only prevent some rules from matching, but otherwise do not change the way they work, the technique is sound, too.

This heuristic was developed to make inductive proofs more tractable. Inductive proofs play a very important role in program analysis, and one can easily imagine advanced transformation systems that exploit program equivalences which require such proofs.

⁴This heuristic is presented in [BSvH⁺93].

⁵So-called wave-rules.

5.2 Giving priority to choices

Another way to improve efficiency is to give priority to transformations based on some heuristic measure for them. Such a measure may be the size of certain subprograms that we want to transform or the “distance” between interesting subterms in the abstract syntax tree. There should, of course, be some kind of statistical motivation behind such measures that makes it more likely to apply the right transformations to the program first.

For example, when having the choice between unfolding any of a number of recursive function calls, the question may be, which of them is more likely to yield a program which we can further simplify? One such heuristic may be to unfold the outermost recursive function first: because it encompasses a larger part of the program, the partial evaluator may have more opportunities to optimise, because the function parameter might get substituted in many more positions.

Another heuristic would be to prefer recursive function calls that take a value of sum type as argument: this passes information (choice) from terms higher in the tree to deeper ones, possibly giving rise to more simplifications again. We could even define the following measure for the parameter if it represents a nested datastructure containing sum types: it is more likely that a value of sum type which is higher in the datastructure is consulted than one very deep within the datastructure. Thus, trying function applications with such arguments earlier may be beneficial. E.g.:

$$(*, (\text{inl}(v), *))$$

would be less likely to yield information to the function body than

$$(\text{inl}(v), *)$$

The rationale behind this heuristic is that to reach a value that is deep within a datastructure we need more code (functions that destruct the datastructure

up to the point of the value). Because we want to perform as little work as possible, we might prefer to transform a program part where information is consumed at an earlier step of execution.

5.3 Control and completeness

One aspect that should not be ignored, though it is not relevant to efficiency of the transformation system, is the question of completeness. The search for suitable transformations may in many cases be infinite. What should the system do if it has not found the goal of a transformation method after many iterations? The control part of the system will therefore have to put a reasonable limit on the depth of search so as not to try transformations for an indefinite amount of time.

5.4 Examples of control heuristics

This section will present two control strategies: the classic one by Burstall and Darlington and a still untried one that we believe to be useful in combination with our partial evaluator. The reader might be interested in implementing them using our framework as basis.

5.4.1 A classic control strategy

In their initial paper, Burstall and Darlington propose a useful control strategy that may be applied together with their fold/unfold method. The typical application of the rules of this strategy looks as follows:

1. Use the definition rule to create necessary definitions of recursive equations (functions).
2. Instantiate the equations using the instantiation rule.

3. For each instantiation unfold repeatedly and at each step:
 - Try to apply the algebraic replacement rule and the one for where-abstraction.
 - Try folding.

Unfortunately, in their original system the first two steps require help from the user⁶. The algebraic replacement and where-abstraction rules could be applied automatically, but the system may need the guidance of the user to find solutions faster (due to possibly many alternatives).

Therefore, it may be useful to allow (but not require) user interaction with the system to control rule applications. This might help the user to discover more general principles in guiding the transformation system, which could lead to better automation of the same.

5.4.2 A control strategy with partial evaluation

Since our partial evaluator will never degrade the efficiency of a program and is capable of reducing the number of choice points in the search space as described in this chapter, the first step in this control strategy is to run the partial evaluator on the whole program, followed by the elimination of common computations (sub-expressions). Then we try the following steps:

1. Find next recursion that takes a sum type as argument.
2. Choice point:
 - If the argument of the function is an explicit value of sum type, proceed to next step, otherwise specialise it (i.e. apply the instantiation rule) before as described in our chapter on transformations so as to get such an explicit value.

⁶Discovery of “Eureka”-steps.

- Otherwise fail. This backtracks to step one and tries the next recursive function.
3. Unfold the recursive call — this makes it a lambda abstraction.
 4. Partially evaluate program and eliminate redundant computations: this will apply the newly created lambda abstraction to the explicit values of sum type.
 5. Try folding. If this succeeds it depends on what we want to do: we may continue improving the program with this strategy or we could succeed with the transformed one. Otherwise we have two choices again:
 - Continue transformation recursively at start with next recursive function (the current function stays otherwise transformed as up to this point!).
 - Backtrack to step one, undoing the transformation of the last recursive function.

If we restrict the search space by a depth limit, this strategy will try out all combinations of unfolding recursion up to this depth, possibly specialising function parameters as required. This may also be combined with other rules if this seems useful (e.g. the algebraic replacement rule, some kind of tupling strategy, etc.). The use of the partial evaluator in this strategy guarantees that we will only try folding at points where folding has a chance to succeed. Without it, folding might fail even if the functions only differ by one statically computable expression in a critical place, which can happen all too easily. Additionally, partial evaluation may eliminate code parts that are not needed any more. If these contain calls to recursive functions, our search space will be made smaller without losing any existing goals.

Chapter 6

Implementing Transformations

This chapter will give an introduction to the techniques we used to implement a framework for doing program transformations. We start out by describing properties of the language LambdaProlog, which is a very recent development and highly suitable for tackling problems in the field of program transformation. This will be followed by interesting design techniques that allow us to achieve a high degree of declarativity and impose an easily extensible structure when implementing such a system. An overview of the components implemented in the system is given in appendix B.

6.1 Implementation in LambdaProlog

The recently developed logic programming language *LambdaProlog* is an extension of its cousin *Prolog*: whereas Prolog is a typical example of a logic language that implements first-order logic in terms of Horn clauses, LambdaProlog takes us to a higher level: higher-order logic. This allows us to reason about even predicates and functions, not only first-order terms. This capability already indicates its usefulness for the problem at hand, because in functional languages programs correspond to functions. Hence, implementing program transformation systems is greatly simplified by such powerful

features¹. Here is a short overview of the extensions and advantages which LambdaProlog² has over traditional Prolog:

- Higher-order predicates and functions: functions and predicates can be bound to variables.
- Higher-order unification³: we can unify not only first-order terms but also functions and predicates.
- Explicit universal and existential quantification: we can introduce a scope for universally and existentially⁴ quantified variables. Universally quantified variables can be used in goals, too, and are very useful when reasoning about functions. Universal quantification is introduced with the keyword `pi` followed by the name of the variable and a backslash. The keyword `sigma` is used to scope existential quantification.
- Intuitionistic implication: it is possible to try solving goals that depend on some statement which can be asserted at runtime.
- Static typing: especially helpful, because the higher-order features make it more difficult to write type correct programs.
- Module system: supports software development “in the large”.

¹A general introduction to transforming programs and formulas in LambdaProlog is given in [MN87]. [Han98] also turned out to be a very useful introductory tutorial.

²We mainly used the LambdaProlog implementation “Teyjus”, which is still under development but already sufficiently mature. It is currently available in the WWW at the URL: <http://teyjus.cs.umn.edu>

³It should be noted that higher-order unification is generally undecidable. Thus, every sound implementation is necessarily incomplete: one can always find cases where a more general unifier exists than some unification algorithm finds. In restricted cases, however, (e.g. higher-order patterns - see [DHKP96]) higher-order unification is decidable.

⁴Existential quantification can also be used implicitly when no scope is given.

The language is otherwise similar to Prolog: first-order unification is a special case of higher-order unification, backtracking gives non-deterministic choice between solutions, it has negation-as-failure and cuts⁵.

6.1.1 Higher-order logic

LambdaProlog has its theoretic foundations in an extended version of so-called higher-order hereditary Harrop formulas⁶. Semantically speaking, it is a true super-set of first-order Horn clauses with additional higher-order capabilities, which we will now describe in more detail:

Higher-order predicates and functions

Since they are part of the language design, we have a very natural way of expressing lambda abstractions. For example:

$$(x \backslash \text{pair } x \ x)$$

This is an anonymous function that takes a parameter (called x inside the function body) and builds a term out of it (a pair in this case). We can apply such functions to arguments by juxtaposition:

$$(x \backslash \text{pair } x \ x) (y \backslash \text{inl } y)$$

which evaluates by β -reduction to:

$$\text{pair } (y \backslash \text{inl } y) (y \backslash \text{inl } y)$$

As we can see, even higher-order functions are supported directly.

An example of higher-order predicates:

⁵This unfortunately also means that LambdaProlog is an *impure* logic language as opposed to, for example, Mercury.

⁶See [NM95] for a detailed overview of the semantics of LambdaProlog.

```

type map (A -> B -> o) -> list A -> list B -> o.
map P nil nil.
map P (H1::T1) (H2::T2) :- P H1 H2, map P T1 T2.

```

This predicate `map` maps a predicate over a list of elements. This allows `P` to be a higher-order predicate. If we consider the following knowledge base:

```

type parent string -> string -> o.
parent "Albert" "Alan".
parent "Albert" "Ann".
parent "Berta" "Bill".
parent "Berta" "Bridget".

```

then the variable `Parents` in the following piece of code:

```

mapP parent ["Albert", "Berta"] Parents

```

could be instantiated to any of the following:

```

["Alan", "Bill"]
["Alan", "Bridget"]
["Ann", "Bill"]
["Ann", "Bridget"]

```

We will make very intense use of higher-order predicates when we introduce monads as an implementation technique in section 6.2.

Higher-order unification

This is probably the most powerful aspect of LambdaProlog: it allows instantiating variables that stand for (or contain in substructures) functions and predicates by unifying them with other higher-order terms. To give a short example in which one might be interested when doing program transformation (we use the language representation on which we do transformations):

```

curry (lam (x\ F (fst x) (snd x)))
      (lam (x1\ lam (x2\ F x1 x2))).

```

This is the so-called *Curry-relation*, which relates functions that take pairs as arguments to ones that are *curried*, i.e. functions that take one argument and return a function which again consumes the second argument (this allows more flexible use of higher-order functions).

The “magic” happens with variable **F**: it is a higher-order variable and stands for any kind of program (function) that takes the required arguments (in this example arguments that are pairs).

An example application is:

```
curry X (lam (x1\ lam (x2\ x1))).
```

The answer substitution:

```
X = lam (x\ fst x)
```

Universal quantification

To get an even better impression of the power of higher-order unification, here is another example that demonstrates this, and also universal quantification (the keyword **pi** introduces a scope for a universally quantified variable that it receives as argument — **x** in this case):

```
contains T ST :- T = F ST, not (pi x\ F x = T).
```

The type of this predicate is $A \rightarrow B \rightarrow o$, which means that both **T** and **ST** can have arbitrary type. What the predicate does is that it checks whether some term **T** contains a subterm **ST** — it does not matter whether we are talking about lists, binary trees or any other kind of datastructure. Taking a look at the code, we see that first we declare that some function **F** maps the subterm to the term⁷. However, **F** could be a trivial function that always returns **T** without considering its argument **ST**. Therefore, we enforce that **F**

⁷For clarification: “functions” in LambdaProlog only allow simple term substitution. There is no term evaluation of any kind happening when we try to infer a higher-order variable (here: a function).

depends on its parameter by declaring that “Not for all x may $(F\ x)$ yield T ”. This can only succeed if there is an x such that it appears in T — in other terms: ST must be contained in it as required.

It should be noted that ST need not be instantiated — it can be a variable. This would allow us to find out about all subterms contained in T (ST would be instantiated to them by unification and backtracking).

Intuitionistic implication

We take another example from the implementation of our system:

```
infer_tp (lam F) (TP1 --> TP2) :-
  pi x\ infer_tp x TP1 => infer_tp (F x) TP2.
```

The purpose of predicate `infer_tp` is to infer the type of terms in our language. Terms are given as first argument, and the second argument of the predicate will be instantiated to the type of the term. The rule given here infers the type of lambda abstractions. What the body of this rule is saying is that “For all x , inferring type $TP1$ for x implies that the return type of $(F\ x)$ can be inferred as $TP2$ ”. This means that if the inference system requires the type of x in the function body of F during inference, it will get $TP1$. This type may, of course, be further constrained depending on the way x is used in the function (e.g. as a pair). If its use conflicts with the type as inferred so far, the predicate will fail, thus indicating a type error. We see again that LambdaProlog allows us to specify important properties of programs in a very clear and concise way.

6.2 Structuring partial evaluation using monads

As we have described in chapter 4, partial evaluation in strict languages is faced with tricky correctness issues concerning non-termination and other

side effects. Since the core of our framework is a partial evaluator, it is necessary to give a detailed treatment of the design technique applied. We will see that this specific technique achieves the following:

- It solves the problems concerning correctness in a most elegant way: partial evaluation is guaranteed to preserve the side effects of programs.
- It allows easy extension to cover side effects that are not currently handled by our partial evaluator.
- The implementation is highly declarative, which should facilitate correctness proofs.
- The partial evaluator is very efficient.
- Last but not least it builds on strong theoretic foundations.

Many different designs had been tried to come up with a general and clear implementation of this component, but the tricky correctness issues concerning side effects made this fairly difficult. It required quite some time until the relations between several concepts that might be applicable became clear. Although knowledge of work concerning handling of side effects in purely functional languages⁸ was available from the start of the project, the linking idea appeared after having read [BT95]: this paper shows that even logic languages as expressive as LambdaProlog can strongly benefit from a specific construct whose introduction to functional languages has opened new ways of programming: *monads*.

⁸The reader may find it very beneficial to read [Wad92] and further papers like [JW93] and [Wad97].

6.2.1 What are monads?

Monads arose in category theory and their usefulness in structuring denotational semantics was discovered by Eugenio Moggi⁹. Wadler took up this idea and applied it to structuring functional programs. Other people had also noted that monads are a powerful tool for encapsulating the handling of side effects (e.g. exceptions) in a safe and referentially transparent way in functional languages.

Monads allow us to reason about computations in a very abstract way. This is exactly what we need for our purposes, because computations are the food of partial evaluation. The combination of the ideas presented in the works above lead to an implementation of a monadic approach to partial evaluation. We will not give a full treatment of monads as this is already presented in the papers mentioned. In short, the minimum specification of a monad is a triple of a unary monad type constructor, a unit-operator and a bind-operator. For example, a monad type constructor `m` would impose the following types on the operators:

```
kind m type -> type.
type unitM A -> m A -> o.
type bindM m A -> (A -> m B -> o) -> m B -> o.
```

This is the abstract view of monads: we operate on it using the given operators. The unit-operator lifts some value into a monad, the bind-operator binds a function (in LambdaProlog: predicate) to a value that it gets from the monad and returns a new monad. The monad, however, has a (hidden) internal representation: this representation may carry additional information about the values it controls. This can be, for example, the number of applications of the bind-operator, errors that happened during some monad operation or (as in our case) a classification of terms into values, terminating

⁹See [Mog89] and [Mog91] for details.

computations, etc. One can imagine many applications (see [Wad92]). Because we operate on the monad using the operators only, our code becomes independent of the monad representation: we could add further details to it without breaking code¹⁰ outside, for example to support more kinds of side effects in case the language is being extended.

6.2.2 Using monads to hide side effects

Our specific monad handles computational effects “behind the scenes” (internally). It can do so, because every time a term is put under its control, it will be told what kind of term it is: a value, some (irreducible) computation or a possibly non-terminating computation. For this purpose we use more than one unit-operator, which does not violate the properties of monads (`effM` stands for “effect monad“):

- `unit_value_effM` — when the result of partial evaluation is a value.
- `unit_comp_effM` — in the case of a terminating computation.
- `unit_mnont_effM` — when the result of partial evaluation is a computation that maybe causes non-termination (“mnont”). This is generally undecidable, hence the “maybe”.

They all have the type `A -> effM A -> o` as required by monads. This is interesting to note — the monad does not have the slightest “idea” of how terms are implemented in our language: hence the general type `A` which the monad type constructor accepts as argument. This would allow us to reuse the implementation of the monad to handle effects for any kind of representation of a strict functional language: as we see, an indeed very abstract view¹¹.

¹⁰See appendix C.1 for a full implementation example.

¹¹We will learn in section 6.3 that it is indeed reused for a second implementation of the partial evaluator which maintains type information.

6.2.3 Partial evaluation and monads

Instead of partially evaluating terms to terms, we partially evaluate terms within the monad internally. But how can we get at terms that result from computation in the monad when they are required as subterms in some simplification rule? This can be done with the monadic bind-operator: it takes a monad as first and a (higher-order!) predicate as second argument which “binds” to whatever term the monad finds suitable to “leak” to this predicate. The type of the bind-operator `bind_effM` is:

```
effM A -> (A -> effM B -> o) -> effM B -> o
```

So the bound predicate must instantiate its second argument to a new monad. This means that computations that can be statically evaluated all happen “within” the monad. Here is an example rule of the partial evaluator. Its type is, of course, specific to terms (type `'tm'`), but the monad can handle all types as we mentioned further above (`'tm'` is a specific instance):

```
type part_evalM tm -> tm effM -> o.
part_evalM (fst T) Res :-
  part_evalM T M,
  bind_effM M simplify_fst Res.
```

This predicate takes a term as first argument and returns a monad in the second one which contains the partially evaluated computation. This specific rule handles partial evaluation of the pair accessor `fst`. To do this, we first partially evaluate its parameter within a monad `M`, and then we “bind” a simplification function to whatever the monad allows us to see: this could be a normal pair (which we could simplify then), but also just a universally quantified variable that stands for all possible terms that could be in this place. This would indicate that either there was no computation that could be simplified as e.g. in `lam (x\ fst x)`, where no simplification is possible.

Or it could be the case that the monad “lifted away” a computation that might cause a side effect or represents another computation (not a value). If it did not do so, a simplification rule might accidentally remove this side effect or “multiply” a computation, which can be costly¹². The simplification rules for `fst` look as follows:

```

type simplify_fst tm -> tm effM -> o.
simplify_fst (pair V1 _V2) Res :-
  !, unit_value_effM V1 Res.
simplify_fst V Res :-
  unit_comp_effM (fst V) Res.

```

Either the value to be simplified is a pair, in which case we just drop its second argument (it is a value, of course, as guaranteed by the monad implementation) and only return the first one, advising the monad that this is a value. Or the term is something else (only a universally quantified variable would make sense here). Then we cannot simplify, so return the computation that takes the first element of the pair: `fst V`. The monad learns that this is a computation through the corresponding unit-operator `unit_comp_effM`.

The implementation of the bind-operator for the monad acts accordingly depending on the kind of term it gets. If it is a value, the term will be passed unchanged to the bound predicate. Otherwise, the terminating or maybe non-terminating computation is lifted out of its place and a universally quantified variable is left there instead. If it was a “normal” computation and after the bound predicate finishes simplification, the monad would check whether the universally quantified variable is still present in the simplified program: if this is not the case (a simplification rule must have removed it then), the computation is not needed to compute the result of the program —

¹²This means that our monad helps us maintain the invariant that everything that leaks out of it to bound predicates is a value. Thus, it also enforces in a most natural way that nonlinear functions never get computations as arguments, which could violate linearity restrictions that incur loss of efficiency: computations are never duplicated.

the monad will discard it. Otherwise or when we have a potentially diverging term, the monad will remember the computations in the right order so that no effect is lost.

Of course, at some point of time we will want to get at the result of partial evaluation. For this we provide a `show_effM`-function that reveals the term controlled by the monad: it translates the internal representation of the monad, its constructors used for lifting out computations with 'let'-constructs¹³ as we described in chapter 4.

Here is a final example of a very crucial component of the partial evaluator that demonstrates within a couple of lines only the elegance of the monadic approach and nearly all of the power of LambdaProlog (explicit universal and existential quantification, intuitionistic implication, higher-order functions and higher-order unification):

```
part_eval_fun F1 F2 :-
  pi x\ sigma Mx\
    unit_value_effM x Mx,
  part_evalM x Mx =>
    sigma M\
      part_evalM (F1 x) M,
      show_effM M (F2 x).
```

The purpose of this predicate is to partially evaluate a function from terms to terms (`tm -> tm`), `F1` being the function to be improved, `F2` the result. We already know that in our strict language all variables stand for values. Thus, no matter to which term we apply the function to be partially evaluated, this term must be a value in its body. In LambdaProlog this is expressed as follows:

- “For all terms `x`” (`pi x\`) there “exists a monad `Mx`” (`sigma Mx\`) such

¹³See appendix C.2.2 for the implementation.

that this x can be lifted into the monad as a value (in LambdaProlog: `unit_value_effM x Mx`).

- Partially evaluating x in monad Mx implies (`part_evalM x Mx =>`) that there is another monad M (`sigma M\`) such that we can partially evaluate $F1$ applied to x within this monad M (`part_evalM (F1 x) M`).
- Partially evaluating the statically computable computations which are controlled by monad M results in another term: this last term can be gained by applying the partially evaluated function $F2$ to x (in LambdaProlog: `show_effM M (F2 x)`). In this last part the higher-order variable $F2$ (it stands for a function) is inferred by higher-order unification.

As we have hopefully managed to demonstrate, the advanced features of LambdaProlog combined with the monadic approach result in a most declarative implementation of what it means to partially evaluate a function that is strict in its argument. Assuming that the monad implementation indeed behaves as we have explained further above, this piece of code could be proved correct easily: its declarative reading should actually correspond to the specification itself.

6.2.4 Extensibility and monads

Currently, the monad only handles non-termination and “irreducible computations”¹⁴. It would be very easy, however, to extend its capabilities to handling all kinds of side effects, such as I/O and exceptions. We would not have to change a single rule of the partial evaluator other than the one for function application (which is the only place where effects can be triggered). The monad would have to be extended with more unit-operators that indicate the kind of effect, and the implementation of the binding-operators

¹⁴If the value of a variable cannot be known before runtime, then we cannot know the result of computations: terms containing this variable cannot be fully reduced.

would need to handle those effects accordingly. This pinpoints the place where extensions can happen and gives us a very modular approach to structuring partial evaluation. The big advantages of the monad approach can also be seen in code size: the partial evaluator only requires somewhere between 150 and 200 lines of code¹⁵, effect handling in the monad itself about 30. The level of declarativity is extremely high, and it therefore should not be too difficult to give a proof of correctness of the partial evaluator by just taking the implementation as guideline.

Another aspect of extensibility of our partial evaluator, though not directly related to monads, is that we can arbitrarily extend its power to infer termination behaviour: the partial evaluator makes use of a predicate `funcall_terminates`, which takes a term as first argument that stands for a function and another term as second argument which is the parameter that should be applied to the function. If calling the predicate succeeds, this should mean that calling the given function on the given parameter terminates. Currently, we do not provide any implementation for this predicate, but users who want to try out advanced tools for termination analysis can “hook” them into our system by just using this interface. LambdaProlog allows this very conveniently due to having the property that predicates can be implemented incrementally, spanning their implementation over arbitrary modules.

6.2.5 Theoretical aspects of monads and partial evaluation

It was only after the implementation of the partial evaluator that it became evident that other researchers had already given a full formal treatment of partial evaluation under monads. The interested reader can find such for-

¹⁵The interested reader may want to take a look at the code example in appendix C.

malisations in [HD97a] and [Hat98]. This should be an indication that our approach has a strong formal basis.

Most advantages of having a partial evaluator that identifies various sorts of computation have already been described in our chapter on transformations, but it is interesting to note that the way in which computations are lifted in this implementation with 'let'-constructs has other advantages, too. As, for example, [JPS96] explains, some ways of using 'let' make it easier for compilers to generate fast code. Indeed, programs transformed by our partial evaluator end up in this form automatically: the computation steps, all of which are lifted out of place, follow each other in a linear fashion.

6.2.6 Final remarks on the use of monads

As a final remark on the monadic approach, it should be pointed out that this is a generic technique for structuring declarative code: it can potentially be used for many kinds of problems, e.g. where parts of the code need the guarantee of certain invariants, which the monad can enforce. It usually does this in such a way that changing the way invariants are maintained does not interfere at all with the rest of the code. This seems to make this approach very suitable for most kinds of transformation and rewrite systems in general. One should not forget, however, that monads require higher-order capabilities (higher-order functions and/or higher-order predicates) as they are provided in most functional and in higher-order logic languages. Their safe application usually requires a static type system that allows hiding of implementation details.

6.3 Maintaining types during partial evaluation

There is another implementation of a partial evaluator available in our framework which maintains type annotations during partial evaluation. The rationale behind it is that fold/unfold techniques and possibly others require type information to work correctly and/or efficiently. For example, when we want to apply the instantiation rule to specialise a call to a function, we must make sure that this call indeed takes an argument that contains a sum type in its type signature. Unfortunately, it is very costly to infer this information whenever it is needed: we would have to infer types for the whole program, which is unacceptable. Therefore, we convert terms to another representation which contains type annotations. This is a one-time effort only. The alternative partial evaluator respects these annotations and therefore allows other transformations to find out the exact type of every node in the abstract syntax tree without having to use type inference. Of course, other transformations must respect the type annotations, too.

Chapter 7

System Evaluation

This chapter will present example applications of the most interesting components of our framework. Most examples were tried under the Teyjus toplevel, but they could be easily executed from within a main module, as is also contained in the distribution. The example program will always be bound by existential quantification so as to prevent the toplevel from printing it in the solution part. The application of the predicate of interest follows this definition immediately, then the system output is given. Each example will be explained in detail.

7.1 Type inference

According to the typing rules in chapter 2, our framework allows type inference for a large class of programs, unfortunately not (yet) completely for recursive types. It also contains functionality for type checking polymorphic programs. Here is an example of mixed type inference / type checking of a polymorphically typed program:

```

sigma Program\
  Program =
    tlet
      (all (a\ in (a --> a))) (tabs (a\ tlam a (y\ y)))
      (f\ pair (app f (pair u u)) (app f u)),
    infer_tp Program TP.

```

The answer substitution:

```
TP = (unit ** unit) ** unit
```

As can be seen in the last line of code, the function `f` is applied twice: once to a pair of unit values, then to only the unit value. This is an example of so-called let-polymorphism, which the system can handle due to a few small extensions presented in [Han98].

7.2 Evaluation

This demonstrates the application of the rules of the structural operational semantics as given on page 27.

```

sigma Program\
  Program = snd (pair u (lam (x\ snd (pair u u))))),
  eval Program Result.

```

The answer substitution:

```
Result = lam (x\ snd (pair u u))
```

As we can see clearly, normal evaluation is not capable of simplifying all parts of the program: the body of the resulting lambda abstraction could obviously be improved (reduction of the application of the pair accessor `snd`), but the `eval`-predicate cannot step into the body of functions: this is the application domain of the partial evaluator.

7.3 Partial evaluation

Taking the same example as before for the evaluation rule, we see that partial evaluation can achieve more:

```
sigma Program\  
  Program = snd (pair u (lam (x\  
    snd (pair u u))))),  
  part_eval Program Result.
```

The answer substitution:

```
Result = lam (W1\  
  u)
```

Due to the importance of the partial evaluator in this project, we will give a thorough overview of its capabilities.

Elimination of redundant pair constructions

```
sigma Program\  
  Program = lam (x\  
    pair (fst x) (snd x)),  
  part_eval Program Result.
```

The answer substitution:

```
Result = lam (W1\  
  W1)
```

As can be seen, the program above destructs a pair into its components using the `fst` and `snd` accessors and rebuilds it again with the `pair` constructor. This is redundant and will always be eliminated. Of course, this works in much more complex examples, too, where some resulting datastructure may contain parts which are for some reason reconstructed in this way. The rule is safe, because the accessors always get values only as is ensured by the monad: effect behaviour will remain identical.

Reduction of accessors and the representation operator

We will not give any example for this, because this trivial simplification was already demonstrated in earlier ones. When given a pair at binding time, `fst` and `snd` will immediately reduce it. So will `rep_rtp` when given an abstracted value of recursive type.

Simplification of case statements

Values of sum type constructed by `inl` and `inr` cannot be simplified by themselves, but case statements can — even in many ways! It is trivial that a case statement will be simplified if the outermost constructor of the condition (of sum type) is known. This will reduce the whole case statement to the appropriate case arm. There are, however, two further simplifications:

```
sigma Program\  
  Program =  
    lam (x\  
      case x  
        (l\  
          inr l)  
        (r\  
          inr r)),  
  part_eval Program Result.
```

The answer substitution:

```
Result =  
  lam (W1\  
    let_comp (case W1 (W2\  
      inr W2)  
      (W2\  
        W1))  
    (W2\  
      W2))
```

The result, in which the terminating computation of the case statement could be inlined now (see further below in this chapter), shows that the second case arm has been changed — but why? The answer is similar

to redundant pair-reconstruction: the condition is destructed during the choice of the case arm, only the parameter of its constructor will be bound within the case arm. But `inr` reconstructs it again! Such reconstructions are spotted by the partial evaluator: the condition will be substituted for all occurrences of `inl l` in the first case arm and all occurrences of `inr r` in the second case arm. Changing the above program only slightly (actually: a single letter) shows us the third simplification rule for case statements:

```
sigma Program\  
  Program =  
    lam (x\  
      case x  
        (l\  
          inl l)  
        (r\  
          inr r)),  
  part_eval Program Result.
```

The answer substitution:

```
Result = lam (W1\  
  W1)
```

This may be surprising at first, but is easily understandable: first the partial evaluator discovers that the condition is reconstructed in both case arms. Then it applies its third rule: when two case arms are structurally equivalent, the whole case statement can be eliminated, simply because its result does not depend on the condition of the case statement! This allows it to reduce the example to the identity function. Note again that the condition of the case statement is always a value, which is guaranteed by the monad. Therefore, all these rules are sound: equivalent values can always be substituted for each other without changing the meaning of the program including effect behaviour (this is called referential transparency).

Elimination of “faked” recursion

In some cases the partial evaluator can detect that a recursive function is guaranteed to terminate. This happens when evaluation will never reach a recursive call. Here is an example:

```
sigma Program\  
  Program =  
    app (  
      lam (x\  
        rec (f\  
          case x  
            (l\  
              (r\  
                app f y))))  
      (inl u),  
    part_eval Program Result.
```

The answer substitution:

```
Result = lam (W1\  
  u)
```

The partial evaluator sees that the application of the lambda abstraction in this example is possible, which causes `x` to be substituted in the body of the recursive function. This again allows the partial evaluator to simplify the case statement, which removes the recursive call that is in the eliminated second arm. Finally, it turns out that the recursion variable `f` is not used any more in the body of the recursive function. Therefore it is safe to rewrite the recursive function to a normal lambda abstraction. This might allow application of this resulting function to another value immediately (e.g. when this result is used as a higher-order function in an argument to another function).

Handling of function application

We will not give any examples for this rule since it has been used in previous examples already. The only cases when we can reduce a function application happen when the function is a lambda abstraction (guaranteed to terminate) or if it is recursive and some external termination analyser, which can be “plugged” into the system, proves its termination. This guarantees that the partial evaluator always terminates and that its power only depends on the crucial predicate `funcall_terminates`, which has been described in our chapter on implementation.

7.4 Common sub-expression elimination

We consider the following example:

```
sigma Program\  
  Program =  
    lam (x\  
      pair  
        (fst x)  
        (lam (y\  
          fst x))),  
  part_eval Program Result1.
```

The answer substitution:

```
Result1 =  
  lam (W1\  
    let_comp  
      (fst W1)  
      (W2\  
        pair W2  
          (lam (W3\  
            let_comp (fst W1)  
              (W4\  
                W4))))))
```

The output is somewhat longer, but it should still be possible to see that the pattern “`let_comp (fst W1) ...`” appears twice. Although the second occurrence happens to be in a deeper lambda abstraction where it is not guaranteed to be called if the outermost function is applied, it does not hurt (and maybe help) to eliminate this duplicated computation using the corresponding module for this (see appendix B). When applied to the result of the last example (`elim_cse Result1 Result2`), the output will be:

The answer substitution:

Result2 =

```
lam (W1\  
  let_comp (fst W1)  
            (W2\  
              pair W2 (lam (W3\  
                            W2))))
```

As can be seen, the redundant computation has been eliminated. This is always safe, because this operates on lifted computations without side effects only.

7.5 Inlining

This feature, implemented in module “let_ext”, makes it possible to inline terminating lifted computations again if this does not duplicate them in a nonlinear way (each case arm is considered separately). This would rule out the last example, but improve the one presented in the section on case statements. This program:

```
lam (W1\  
  let_comp (case W1 (W2\  
                    inr W2)  
            (W2\  
              W1))  
          (W2\  
            W2))
```

could then be changed to:

```
lam (W1\  
  case W1 (W2\  
          inr W2) (W2\  
                  W1))
```


Chapter 8

Conclusion

One of the biggest problems during this work was the lack of research in transforming strict functional languages. This is not without reason: as we have seen in many parts of this work, transformations may have unexpected consequences on termination behaviour and other side effects. It seems that most researchers avoid these somewhat annoying problems in order to focus more on the transformation aspects rather than on these “secondary” issues: they choose lazy languages.

8.1 A better partial evaluator?

We have to admit that solving at least some of the covered problems in a clean and general way prevented us from covering more interesting fields of automated program transformation in the detail they would have deserved. Still, it seems that the “side effects” caused by our work, the implementation of a partial evaluator that can cope with higher-order functions and side effects easily while allowing many optimisations and retaining a very elegant implementation, was worth the effort. This is even more true if we consider that other important transformations like common sub-expression elimination or functionality for program analysis like termination inference

greatly benefit from this approach: “squeezing out” all statically computable parts from a program reveals a huge amount of information that we can exploit. One could imagine, for example, improving an extended version of this system that conducts exception analysis to find places where uncaught exceptions could be raised — an application in the field of software verification rather than transformation. A powerful and correct partial evaluator seems to be useful for much more than “just” partial evaluation.

How powerful is our monadic approach to partial evaluation really? Unfortunately, the tight time constraints and limited scope of this work did not allow us to give a more formal treatment of this approach. Although we have found and presented references to interesting theoretical work on this question (for example [HD97a]), it would be very important to prove our specific partial evaluator correct. We have mentioned formal tools (e.g. denotational semantics) that might be very helpful for this task. Our design decisions, especially the one to stick as close as possible to a language specification that is formally well-defined ([Win93]), might pay off in future work.

From the developer’s intuitive point of view it seems to be justified to claim that the only limitation of this partial evaluator is termination inference, a generally undecidable problem, and that it performs optimally within this limit. So far, it was not possible to find any counter-example to this claim, and the way the partial evaluator evolved was based on thorough considerations:

- How can we make a term a value? — Exploit the strictness of the language and lift all computations that cannot be statically computed or might cause non-termination out of the term: everything else must be a value.
- How can we guarantee that transformations will always operate on values only for the sake of safety? — We use a monad to abstract from

computations: it will never pass anything to a rule if the monad does not “know” that it is a value. If it is not, the monad will make it one using the previous idea.

- How can any kind of “control instance” (here: the monad) know how to treat results of simplifications? — We classify the results into different categories: values, terminating but irreducible computations and possibly non-terminating computations. This gives the monad the information it needs to know what to do. Values will be forwarded to the next rule unchanged, terminating computations will be lifted away so that they cannot be duplicated, and they will be discarded if they are not needed after the optimisation. Non-terminating terms will be remembered and left in the program so as not to lose any side effects.

If we consider the high level of declarativity in the implementation, it should not be too difficult to come up with a formal treatment of this partial evaluator. This might be interesting future work: as we can learn from e.g. chapter 5.5 and 8.8 in [JGS93], partial evaluation of functional programs seems to currently still face many problems, be it side effects, termination of the partial evaluator, linearity constraints for function applications or higher-order functions. We hope that our approach, which is significantly supported by the underlying implementation language LambdaProlog, contributes some solutions to annoying problems in this field. Future work may illuminate more formal aspects.

8.2 Correctness issues

Besides the correctness issues concerning the partial evaluator, we have also tried to develop suitable criteria that allow safe application of other transformations. It seems that some questions concerning correct application of

recursive folding are clearer now: we have identified sum types as crucial aspect and have presented a way which may give a bit more generality in deciding whether recursive folding is safe (see section 4.1.2). It seems that this has not yet been explicitly treated in the literature we know.

8.3 Control issues

This could have been the major point of this thesis, but the idea to use partial evaluation to improve control has lured us away from this topic quite far. It is definitely clear that this aspect is the one which has to be strongly improved before advanced transformations can be applied to real-world problems, in which search space sizes are very often not tractable for our current methods. Our partial evaluator probably adds less to this topic than to others like correctness and transformation itself.

8.4 New ways of implementation

One contribution of this thesis may be seen in the demonstration that moving to a higher-order logic can lead to significant improvements in our application domain: we have hopefully succeeded in convincing the reader of the clarity and conciseness of programs written in the language LambdaProlog. We therefore suggest that this language be more widely applied, especially in the field of program transformation.

Monads as implementation technique have been gaining quite some momentum in functional languages and have been successfully applied to many very diverse problems. It seems likely that they may have a similar success in higher-order logic languages: we can hardly imagine how to “beat” the clarity and conciseness of the monadic approach that lead to our implementation of the partial evaluator — taking out only a single clause would seem

to make it inherently incomplete. It should be noted that several other approaches had been tried by us before, but none came only close to the current solution. This big step forward gives us confidence that this implementation technique may be highly beneficial in many more cases.

8.5 Future work

Having given a broad overview of various transformation techniques, there should be plenty of ideas that the reader may try out in our transformation framework: on the practical side the fold/unfold strategy is surely a worthy candidate for implementation. But also theoretical work, especially what concerns correctness proofs and improvements in controlling search, could help make this framework even better.

8.6 Completely different ways

Due to the size of the topic, it was not possible to cover alternative approaches to program transformation in any detail. We would like to point out two publications which seem to go fundamentally different ways to automated transformation than we have described so far. The interested reader may therefore consider [Bel94] and [CDPR97]. The first publication approaches program transformations from the field of rewrite rules and completion techniques. The second one takes up the idea of attribute grammars to transforming declarative programs.

Who knows, the answer to many of our questions may be found in approaches that we have considered least promising so far. . .

Bibliography

- [BD77] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
- [Bel94] Francoise Bellegarde. Automating synthesis by completion. Technical Report CS/E 94-20, Oregon Graduate Institute Of Science & Technology, Department of Computer Science and Engineering, April 1994.
- [BSvH⁺93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, August 1993.
- [BT95] Yves Bekkers and Paul Tarau. Monadic constructs for logic programming. In John Lloyd, editor, *Proceedings of the International Symposium on Logic Programming*, pages 51–65, Cambridge, December 4–7 1995. MIT Press.
- [CDPR97] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Rousset. Attribute grammars and functional programming deforestation. In *Fourth International Static Analysis Symposium – Poster Session*, Paris, France, September 1997.
- [DHKP96] G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Unification via explicit substitutions: The case of higherorder patterns, 1996.

- [Erw99] Martin Erwig. *Grundlagen funktionaler Programmierung*. R. Oldenbourg Verlag, 1999.
- [Han98] John Hannan. Program analysis in lambdaProlog. *Lecture Notes in Computer Science*, 1490:353–??, 1998.
- [Hat98] John Hatcliff. Foundations for partial evaluation of functional programs with computational effects. *ACM Computing Surveys*, 30(3es):??–??, September 1998. Article 13.
- [HD97a] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–541, October 1997.
- [HD97b] John Hatcliff and Olivier Danvy. Thunks and the λ -calculus. *Journal of Functional Programming*, 7(3):303–319, May 1997.
- [JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).
- [JPS96] Simon Peyton Jones, Will Partain, and Andre Santos. Let-floating: Moving bindings to give faster programs. *ACM SIGPLAN Notices*, 31(6):1–12, June 1996.
- [JW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Principles of Programming Languages*, January 93.
- [LS95] John Launchbury and Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Conf. Record 7th ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San*

- Diego, CA, USA, 25–28 June 1995*, pages 314–323. ACM Press, New York, 1995.
- [MH95] Erik Meijer and Graham Hutton. Bananas in space: extending fold and unfold to exponential types. In *Functional Programming & Computer Architecture*, San Diego, US, 1995.
- [MN87] D. Miller and G. Nadathur. A logic programming approach to manipulating formulas and programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 379–388, San Francisco, August - September 1987. IEEE, Computer Society Press.
- [Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991. Proposes using the category-theoretic notion of *monad* as a means of parametrizing programming-language semantics by a notion of value-producing computation. This proposal has been widely taken up in research on adding state constructs to functional programming languages.
- [Mog89] E Moggi. Computational lambda-calculus and monads. In *Proceedings of the Logic in Computer Science Conference*, 89.
- [MS95] L. Michael and I. Schwartzbach. Polymorphic type inference, 1995.
- [NM95] Gopalan Nadathur and Dale Miller. *Higher-order logic programming*, volume 5, chapter ? Oxford University Press, 1995.
- [Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK, 1998.

- [PP96] Alberto Pettorossi and Maurizio Proietti. Rules and strategies for transforming functional and logic programs. *ACMCS*, 28(2):360–414, June 1996.
- [Sim99] Alex Simpson. Lecture notes on formal programming language semantics, 1999. Division of Informatics, University of Edinburgh.
- [Wad90] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990.
- [Wad97] Philip Wadler. How to declare an imperative. *ACMCS*, 29(3):240–263, September 1997.
- [Wad92] Philip Wadler. The essence of functional programming. In *Principles of Programming Languages*, January 92.
- [Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing series. MIT Press, February 1993.

Appendix A

Miscellaneous Examples

A.1 Fibonacci in $O(\log_n)$ — Haskell code

```
fib3 n =
  if n <= 1 then 1
  else
    if 2*k == n then ab*ab + b*b
    else ab*ab + 2*b*ab
  where k = n `div` 2
        (a,b) = aux k
        ab = a + b
        aux 0 = (1,0)
        aux 1 = (0,1)
        aux n =
          if 2*k == n then (aa+bb, ab2bb)
          else (ab2bb, ab2bb+aa+bb)
        where k = n `div` 2
              (a,b) = aux k
              aa = a*a
              bb = b*b
              ab2 = 2*a*b
              ab2bb = ab2+bb
```


Appendix B

Components of the system

We will briefly list the components and features of the implemented framework. The full system documentation will be available from the Department of Artificial Intelligence at the University of Edinburgh. See chapter 7 for some examples of the capabilities of the most important components.

Terms and operational semantics

Two modules, “terms” and “oper_sem”, implement the abstract syntax and the operational semantics (evaluation rules) of our language as described in chapter 2.

Types and typing rules

The modules “types” and “typing” contain the type constructors and the typing rules. See chapter 2 for details again. Please note that we have extended the type system in other modules to allow polymorphic typing. This is not documented in our chapter on semantics — it is only an add-on. The additional polymorphic constructs and rules are contained in modules “poly_terms”, “poly_oper_sem”, “poly_types” and “poly_typing”. Background knowledge on polymorphic type inference can be found in [MS95]. It should also be noted that type checking is

not fully working yet: unfortunately, time was too short to implement a complete algorithm that can handle recursive types in more than just trivial cases.

A monad for effects

The monad we use for handling computational effects as described in chapter 6 is implemented in module “`effect_monad`”.

Partial evaluation

The partial evaluator as presented in chapter 6 can be found in module “`part_eval`”.

Different kinds of “`let`” + inlining

We make use of different kinds of “`let`”-terms so as to reflect the reason why some term has been lifted out of its place. All of them take a term as first argument and as second argument a function from terms to terms in which the first term should be substituted at runtime. This is implemented in module “`let_ext`”.

Besides a general “`let`” that does not make any assumptions about its parameters, there is a “`let_comp`” and a “`let_mnont`”. The latter two originate in the partial evaluator: they replace the monad constructors of similar name (“`unit_comp_effM`” and “`unit_mnont_effM`”). They indicate whether some lifted expression is a terminating computation or maybe a non-terminating one. This information can be used by other transformations or program analysis predicates.

Simple termination inference

This functionality, which is implemented in module “`termination`”, allows the user to check whether evaluating a term will terminate (i.e. whether the term converges to a value). If this succeeds, the term is

guaranteed to terminate, if it fails, it may still terminate but can also cause non-termination (generally undecidable). This component makes use of the information provided by the partial evaluator (the classification of computations into terminating and possibly non-terminating ones).

Elimination of redundant computations

The module named “cse” (common sub-expression elimination), which also requires information provided by the partial evaluator, finds terminating computations that are duplicated in the program and replaces them with just one computation whose result is reused. Since it depends on results of the partial evaluator, its power also seems to depend on the capability to infer termination behaviour only: it should otherwise do a perfect job in the bounds of this limitation.

Partial evaluator with type maintainance

As described in section 6.3, there is an alternative implementation of the partial evaluator which maintains type annotations. This part comprises three modules: “tp_terms”, “tp_part_eval” and “tp_let_ext”. This partial evaluator reuses the monad to handle effects. It can also handle polymorphically typed programs, though there are still some unresolved issues concerning simplification of terms that require let-polymorphism. The current implementation takes a safe approach and does not simplify them. The corresponding rules are marked in the source code with a comment that contains “TODO”.

Utilities

The implementation contains a utility module “utils” with commonly useful functionality, for example to eliminate subterms in Lambda-Prolog terms of arbitrary type. The current distribution also contains a

generic Makefile that builds LambdaProlog projects and a VIM-syntax file for highlighting LambdaProlog source codes in the VIM-editor (Vi-Improved). Additionally, there is a small script “make_terzo.ml” written in OCaml plus an auxiliary LambdaProlog-module “terzo_stuff” which can be used to translate LambdaProlog sources that were written for use with Teyjus to source codes that can be accepted by the Terzo-implementation of LambdaProlog. The latter was very useful when it was not sure whether Teyjus, which was still under heavy development at the time of the project, had bugs or whether the program was incorrect. Comparisons with Terzo helped track down many bugs in the Teyjus-implementation¹.

¹If nothing else, this project has contributed to making Teyjus a much stabler compiler due to a large number of bug reports.

Appendix C

Examples of System Code

Due to space restrictions we only present the crucial components of the system: the part that contains effect handling using monads and the partial evaluator that makes use of it.

The whole framework consists of currently 20 modules that contain about 2000 lines of code (including many comments and empty lines).

The implementation will be available from the Department of Artificial Intelligence at the University of Edinburgh.

C.1 Code of effect monad

C.1.1 Signature of effect monad

```
sig effect_monad.
```

```
% EFFECT MONAD
```

```
% The effect monad guarantees that computational effects are hidden from  
% predicates to which "bind" passes terms: in other terms, these terms  
% are guaranteed to be values. The specific implementation here makes  
% a distinction between values (do not require computation) possibly  
% non-terminating computations and other computations. The reason for  
% the latter is that "computations" that cannot be done statically  
% (reduced to a value) might induce non-linearity if they are applied  
% without restrictions.
```

```
kind effM type -> type. % kind of effect monads
```

```
% MONAD REPRESENTATION
```

```
type value_effM A -> effM A.  
type mnont_effM A -> (A -> effM A) -> effM A.  
type comp_effM A -> (A -> effM A) -> effM A.
```

```
% MONAD OPERATORS
```

```
% [unit_value_effM +T ?M] lifts term [T] to the computation (value of  
% monadic type) [M], indicating that [T] is a value.  
type unit_value_effM A -> effM A -> o.
```

```
% [unit_mnont_effM +T ?M] lifts term [T] to the computation [M],  
% indicating that evaluating it may cause non-termination.  
type unit_mnont_effM A -> effM A -> o.
```

```
% [unit_comp_effM +T ?M] lifts term [T] to the computation [M], indicating  
% that it still requires (terminating) computation to become completely  
% reduced.  
type unit_comp_effM A -> effM A -> o.
```

```
% [bind_effM +MA +P ?MB] binds the computation [MA] to predicate [P],  
% returning the resulting value of monadic type in [MB].  
type bind_effM effM A -> (A -> effM B -> o) -> effM B -> o.
```

```
% ADDITIONAL FUNCTIONS
```

```

% [show_effM +MA ?A] converts the representation of computation [MA]
% to a value [A] of the type on which the effect monad operates. Has to
% be provided by the user!
type show_effM effM A -> A -> o.

% [lifted_term +V ?T] gets the term [T] that has been lifted out of a
% computation using variable [V] in its place.
type lifted_term A -> A -> o.

```

C.1.2 Implementation of effect monad

```

module effect_monad.

unit_value_effM T (value_effM T).
unit_mnont_effM T (mnont_effM T value_effM).
unit_comp_effM T (comp_effM T value_effM).

bind_effM (value_effM T) K Res :- K T Res.

bind_effM (mnont_effM T F1) K (mnont_effM T Res) :-
  pi lt\ sigma R\
    lifted_term lt T =>
      bind_effM (F1 lt) K R,
      ( % CASE: result of binding is a value
        value_effM (F2 lt) = R,
        Res = (t\ value_effM (F2 t))
      ; % CASE: result of binding maybe non-termination or computation
        C (F2 lt) (F3 lt) = R,
        Res = (t\ C (F2 t) (F3 t))
      ).

bind_effM (comp_effM T F1) K Res :-
  pi lt\ sigma R\
    lifted_term lt T =>
      bind_effM (F1 lt) K R,
      ( % CASE: R does not contain lifted term
        Res = R
      ; % CASE: R contains lifted term
        ( % CASE: result of binding is a value
          value_effM (F2 lt) = R,
          Res = comp_effM T (t\ value_effM (F2 t))
        ; % CASE: result of binding maybe non-termination or computation
          C (F2 lt) (F3 lt) = R,
          Res = comp_effM T (t\ C (F2 t) (F3 t))
        )
      ).

```

C.2 Code of monadic partial evaluator

C.2.1 Signature of monadic partial evaluator

```
sig part_eval.

accum_sig let_ext.
accum_sig effect_monad.

% PARTIAL EVALUATION WITH EFFECT ANALYSIS

% [part_eval +T1 ?T2] partially evaluates term [T1] to [T2]. The
% effect behaviour (non-termination, impureness) of [T1] is
% preserved. Side-effecting terms are lifted to the outermost level
% (toplevel or function abstractions). See module "let_ext" for additional
% terms associated with lifting out side effects.
type part_eval tm -> tm -> o.

% [part_eval_fun +F1 ?F2] partially evaluates term function [F1] to
% [F2]. The effect behaviour (non-termination, impureness) of [F1]
% is preserved.
type part_eval_fun (tm -> tm) -> (tm -> tm) -> o.

% [funcall_terminates +F +T] checks whether function [F] (in term
% representation) terminates when applied to term [T]. There are
% no defaults for this, of course, but you may "plug in" your own
% termination analyser.
type funcall_terminates tm -> tm -> o.
```

C.2.2 Implementation of monadic partial evaluator

```
module part_eval.

accumulate utils.
accumulate let_ext.
accumulate effect_monad.

% PART_EVAL

part_eval T1 T2 :-
  part_evalM T1 M,
  show_effM M T2.

% PART_EVALM

type part_evalM tm -> effM tm -> o.
```

```

part_evalM T Res :-
  T = u,
  unit_value_effM T Res.

part_evalM (pair T1 T2) Res :-
  part_evalM T1 M1,
  part_evalM T2 M2,
  bind_effM M1 (V1\ bind_effM M2 (simplify_pair V1)) Res.

part_evalM (fst T) Res :-
  part_evalM T M,
  bind_effM M simplify_fst Res.

part_evalM (snd T) Res :-
  part_evalM T M,
  bind_effM M simplify_snd Res.

part_evalM (inl T) Res :-
  part_evalM T M,
  bind_effM M (V\ unit_value_effM (inl V)) Res.

part_evalM (inr T) Res :-
  part_evalM T M,
  bind_effM M (V\ unit_value_effM (inr V)) Res.

part_evalM (case CT LF RF) Res :-
  part_evalM CT M,
  bind_effM M (CV\ simplify_case CV LF RF) Res.

part_evalM (lam F1) Res :-
  part_eval_fun F1 F2,
  unit_value_effM (lam F2) Res.

part_evalM (rec F1) Res :-
  pi f\ sigma Mf\
    unit_comp_effM f Mf,
  part_evalM f Mf =>
    sigma Fx\
      part_eval_fun (F1 f) Fx,
      ( % CASE: function not recursive anymore -> normal function
        unit_value_effM (lam Fx) Res
      ; % CASE: function still potentially recursive
        F2 f = Fx,
        unit_comp_effM (rec F2) Res
      ).

part_evalM (app T1 T2) Res :-
  part_evalM T1 M1,

```

```

part_evalM T2 M2,
bind_effM M1 (V1\ bind_effM M2 (simplify_app V1)) Res.

part_evalM (abs_rtp T) Res :-
part_evalM T M,
bind_effM M (V\ unit_value_effM (abs_rtp V)) Res.

part_evalM (rep_rtp T) Res :-
part_evalM T M,
bind_effM M simplify_rep_rtp Res.

% PART_EVAL_FUN

part_eval_fun F1 F2 :-
pi x\ sigma Mx\
unit_value_effM x Mx,
part_evalM x Mx =>
sigma M\
part_evalM (F1 x) M,
show_effM M (F2 x).

% SIMPLIFICATION RULES

type simplify_pair tm -> tm -> effM tm -> o.
type simplify_fst tm -> effM tm -> o.
type simplify_snd tm -> effM tm -> o.

type simplify_case tm -> (tm -> tm) -> (tm -> tm) -> effM tm -> o.
type simplify_case_arm tm -> (tm -> tm) -> (tm -> tm) -> (tm -> tm) -> o.
type simplify_case_arms tm -> (tm -> tm) -> (tm -> tm) -> effM tm -> o.

type simplify_app tm -> tm -> effM tm -> o.
type simplify_term_app tm -> tm -> effM tm -> o.

type simplify_rep_rtp tm -> effM tm -> o.

% Pair reconstructs its destructed form -> simplify.
% This is safe, because V, too, is guaranteed to be a value!
simplify_pair V1 V2 Res :-
lifted_term V1 (fst V),
lifted_term V2 (snd V),
!,
unit_value_effM V Res.

simplify_pair V1 V2 Res :- unit_value_effM (pair V1 V2) Res.

simplify_fst (pair V1 _V2) Res :- !, unit_value_effM V1 Res.

```

```

simplify_fst V Res :- unit_comp_effM (fst V) Res.

simplify_snd (pair _V1 V2) Res :- !, unit_value_effM V2 Res.
simplify_snd V Res :- unit_comp_effM (snd V) Res.

simplify_case (inl V) LF _RF Res :- !, part_evalM (LF V) Res.
simplify_case (inr V) _LF RF Res :- !, part_evalM (RF V) Res.

% Condition not reducable to choice.
simplify_case CV LF1 RF1 Res :-
  part_eval_fun LF1 LF2,
  part_eval_fun RF1 RF2,
  simplify_case_arm CV LF2 inl LF3,
  simplify_case_arm CV RF2 inr RF3,
  simplify_case_arms CV LF3 RF3 Res.

% Case arm a function of condition.
simplify_case_arm CV F1 Make F2 :-
  pi x\ sigma Mx\
    unit_value_effM x Mx,
  part_evalM x Mx =>
    sigma F\ sigma M\
      elim_sub_term (F1 x) (Make x) F,
    !,
  part_evalM (F CV) M,
  show_effM M (F2 x).

% Case arm not a function of condition.
simplify_case_arm _CV F _Make F.

% Left and right case arm equivalent.
simplify_case_arms CV LF RF Res :-
  pi l\ pi r\
    EQ = LF l,
    EQ = RF r,
    !,
  part_evalM EQ Res.

% Case arms not equivalent.
simplify_case_arms CV LF RF Res :- unit_comp_effM (case CV LF RF) Res.

% Application of simple lambda abstraction.
simplify_app (lam F) V Res :- !, part_evalM (F V) Res.

% Terminating function applications.
simplify_app FV V Res :-
  funcall_terminates FV V,
  !,
  simplify_term_app FV V Res.

```

```

% Maybe non-terminating function applications.
simplify_app FV V Res :- unit_mnont_effM (app FV V) Res.

% Terminating recursive function application.
simplify_term_app FV V Res :-
  FV = rec F,
  !,
  part_evalM (F FV V) Res.

% Terminating unknown function.
simplify_term_app FV V Res :- unit_comp_effM (app FV V) Res.

simplify_rep_rtp (abs_rtp V) Res :- !, unit_value_effM V Res.
simplify_rep_rtp V Res :- unit_comp_effM (rep_rtp V) Res.

% SHOW_EFFM

show_effM (value_effM T) T.
show_effM (mnont_effM T F) (let_mnont T G) :- pi t \ show_effM (F t) (G t).
show_effM (comp_effM T F) (let_comp T G) :- pi t \ show_effM (F t) (G t).

% SPECIAL RULES FOR LIFTED TERMS

part_evalM LT Res :-
  (
    let T F = LT
  ;
    let_mnont T F = LT
  ;
    let_comp T F = LT
  ),
  part_evalM T M,
  bind_effM M (V \ part_evalM (F V)) Res.

part_evalM LT Res :-
  lifted_term LT _T,
  unit_value_effM LT Res.

```