Basic Research in Computer Science

Topics in Semantics-based Program Manipulation

Bernd Grobauer

BRICS Dissertation Series **DS-01-6**

ISSN 1396-7002 **August 2001**

See back inner page for a list of recent BRICS Dissertation Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `DS/01/6/`

Topics in Semantics-based Program Manipulation

Bernd Grobauer

Ph.D. Dissertation

BRICS

BRICS Ph.D. School
Department of Computer Science
University of Aarhus
Denmark

July 2001

Supervisor: Olivier Danvy

Topics in Semantics-based Program Manipulation

A dissertation
presented to the Faculty of Science
of the University of Aarhus
in partial fulfillment of the requirements for the
Ph.D. degree

by
Bernd Grobauer
31 July 2001

Abstract

Programming is at least as much about manipulating existing code as it is about writing new code. Existing code is *modified*, for example to make inefficient code run faster, or to accommodate for new features when reusing code; existing code is *analyzed*, for example to verify certain program properties, or to use the analysis information for code modifications. Semantics-based program manipulation addresses methods for program modifications and program analyses that are formally defined and therefore can be verified with respect to the programming-language semantics. This dissertation comprises four articles in the field of semantics-based techniques for program manipulation: three articles are about *partial evaluation*, a method for program specialization; the fourth article treats an approach to *automatic cost analysis*.

Partial evaluation optimizes programs by specializing them with respect to parts of their input that are already known: Computations that depend only on known input are carried out during partial evaluation, whereas computations that depend on unknown input give rise to residual code. For example, partially evaluating an interpreter with respect to a program written in the interpreted language yields code that carries out the computations described by that program; partial evaluation is used to remove interpretive overhead. In effect, the partial evaluator serves as a compiler from the interpreted language into the implementation language of the interpreter. Compilation by partial evaluation is known as the first *Futamura projection*. The second and third Futamura projection describe the use of partial evaluation for compiler generation and compiler-generator generation, respectively; both require the partial evaluator that is employed to be *self applicable*.

The first article in this dissertation describes how the second Futamura projection can be achieved for type-directed partial evaluation (TDPE), a relatively recent approach to partial evaluation: We derive an ML implementation of the second Futamura projection for TDPE. Due to the differences between 'traditional', syntax-directed partial evaluation and TDPE, this derivation involves several conceptual and technical steps. These include a suitable formulation of the second Futamura projection and techniques for making TDPE amenable to self-application.

In the second article, compilation by partial evaluation plays a central role for giving a unified approach to goal-directed evaluation, a programming-language paradigm that is built on the notions of backtracking and of generating successive results. Formulating the semantics of a small goal-directed language as a *monadic* semantics—a generic approach to structuring denotational semantics—allows us to relate various possible semantics to each other both conceptually and formally. We thus are able to explain goal-directed evaluation using an intuitive list-based semantics, while using a continuation semantics for semantics-based compilation through partial evaluation. The resulting code is comparable to that produced by an optimized compiler described in the literature.

The third article revisits one of the success stories of partial evaluation, the

generation of efficient string matchers from intuitive but inefficient implementations. The basic idea is that specializing a naive string matcher with respect to a pattern string should result in a matcher that searches a text for this pattern with running time independent of the pattern and linear in the length of the text. In order to succeed with basic partial-evaluation techniques, the naive matcher has to be modified in a non-trivial way, carrying out so-called *binding-time improvements*. We present a step-by-step derivation of a binding-time improved matcher consisting of one problem-dependent step followed by standard binding-time improvements. We also consider several variants of matchers that specialize well, amongst them the first such matcher presented in the literature, and we demonstrate how variants can be derived from each other systematically.

The fourth article is concerned with program analysis rather than program transformation. A challenging goal for program analysis is to extract information about time or space complexity from a program. In complexity analysis, one often establishes *cost recurrences* as an intermediate step, and this step requires an abstraction from data to data size. We use information contained in dependent types to achieve such an abstraction: Dependent ML (DML), a conservative extension of ML, provides dependent types that can be used to associate data with size information, thus describing a possible abstraction. We automatically extract cost recurrences from first-order DML programs, guiding the abstraction from data to data size with information contained in DML type derivations.

Acknowledgments

I wish to thank Peter Dybjer and Neil Jones for serving on my Ph.D. committee.

I am grateful to my Ph.D. supervisor Olivier Danvy for his dedication in supervising his students—the German term for "Ph.D. supervisor" is "Doktor*vater*", which applies to Olivier very much in the literal sense. During my Ph.D. studies, I could be sure of Olivier supporting me all the way. His support was not only scientific but also personal, guiding me and helping me with all aspects of doing a Ph.D. and some besides. He further put much energy and enthusiasm into creating a very special atmosphere in his research group, turning the "Outpost" in the Officersbygning into an ideal place for study and work.

I wish to thank Zhe Yang for sharing his zest for research and life with me: On the same day, we might explore semantics and swing, categories and cappuccino, or monads and music. I am indebted to Zhe for giving me encouragement, advice and help whenever needed. In Zhe I found both a great colleague and a true friend.

I thank Julia Lawall for a rewarding collaboration and for her continued interest into my work: She listened patiently to my sometimes half-baked ideas and read through many manuscripts in various stages of completion, always providing me with substantial comments and encouragement.

For their interest and encouragement I also thank Tobias Nipkow and Gilles Barthe. To Tobias I am further grateful for letting me spend a semester as a research student in his group at Technische Universität München—a both scientifically and personally rewarding visit.

It was a privilege to work and study at BRICS and DAIMI, where I found everybody equally approachable, ready to work together, ready to help. I am grateful to everyone contributing to this environment, and would like to especially thank a few persons: Mogens Nielsen for his encouragement and guidance, and for chairing my Ph.D. committee; Janne Christensen and Karen Møller for giving assistance and a smile whenever needed; Thomas Hune for taking good care of me during the first semester; Paola Quaglia for providing wisdom and optimism; Daniel Damian for many shared hours of working and idling in the O-building; Andrzej Filinski for listening with patience, commenting with substance, and proof-reading with the eyes of an eagle.

Without the love and constant support of my family, I would not have been in a position to start doing a Ph.D., much less to actually complete it. I thank my parents and my sister Bärbel for being a wonderful family.

Finally, I especially thank my girlfriend Alina whose encouragement and support carried me towards completing this thesis.

Contents

Chapter 1

Introduction and Overview

Only few programs are written completely from scratch. To a large extent, programming means manipulating existing code, both for carrying out modifications to the code and for analyzing the behavior of the code.

Code maintenance, for example, requires the modification of existing code: The premises under which some code was written may change, often entailing code modifications. Such changes, to take a few examples, become necessary because of the upgrade to a new platform, the introduction of new standards (consider, e.g., the plethora of standards concerning hypertext markup languages, or the introduction of the euro), or simply the turn of a new century. The latest turn of century caused the so-called *Y2K problem*, forcing large-scale code modifications at a considerable cost: estimates range from US$ 1 to US$ 8.50 per line of code.

Code reuse means the use of existing code within new programs. Reusability of code can be improved by appropriate program design, e.g., as a composition of modules that are as generic as possible and have a well-defined interface; it may pay to go even a step further and form code libraries, which can be reused with particular ease. Also adapting code not written with an eye to future reuse, however, may be cheaper than writing a new program from scratch.

Making a virtue out of necessity, some programming methodologies suggest systematic code modification as a way to overcome a trade-off between clarity and efficiency in programming: A program that implements a straightforward solution to a problem is often rather inefficient, whereas efficient programs tend to be quite cryptic. Systematic code manipulation can be used to start development with a clear but inefficient implementation, which then is gradually rewritten into an efficient one.

Program manipulation is also used to analyze code, for example to find out whether a given program has certain properties. Especially with the widespread use of the Internet, more and more code from non-trustworthy sources run, requiring guarantees about the behavior of the code. A well-known example is the byte code verifier of the Java Virtual Machine, which checks whether programs are well-behaved regarding, e.g., memory access. Other analyses gather

information useful for the programmer, e.g., for debugging purposes or as a basis for program modifications: Program analysis can answer questions such as "Which objects can a given pointer refer to?" and "Is this piece of code actually reachable during execution?"

Program modifications and program analyses should be correct, i.e., modifying a given program should not introduce bugs and program analysis should yield correct results. A proof of correctness, however, requires a formal definition. *Semantics-based program manipulation* addresses formally defined program transformations and program analyses that can be verified with respect to the programming-language semantics.

This dissertation treats topics in semantics-based program manipulation, contributing to the fields of partial evaluation, a transformation for program specialization, and of automated cost analysis. In the remainder of this chapter, we first provide some background on partial evaluation and automated cost analysis, and we describe our contributions. We then give an overview of the articles that constitute the remaining chapters of this thesis. All articles presented here have been peer-reviewed in conference proceedings and/or scientific journals.

1.1 Partial Evaluation

There is much to be said about partial evaluation—here we only sketch some basics, explaining the *what* and giving only a vague idea about the *how*. We then describe how partial evaluation is used in practice, also describing several applications that show how a wide range of useful program modifications can be carried out using partial evaluation. Finally, we describe our contributions to the field of partial evaluation and put them into context.

A complete account of both the concepts of partial evaluation and many of its techniques can be found in Jones, Gomard and Sestoft's textbook [32]. A concise survey can also be found in Consel and Danvy's tutorial notes [13].

1.1.1 Basic concepts

Partial evaluation is a technique for program optimization. It works by specializing programs with respect to parts of their input, the so-called *static* input. This process effectively stages computation: Computations which depend only on the static input are carried out during partial evaluation. At the same time, residual code is produced for computations dependent on *dynamic* input, i.e., input which is unknown at partial-evaluation time. Partial evaluation is carried out automatically by so-called *partial evaluators*.

A Simple Example of Partial Evaluation

The standard example of partial evaluation is specializing the power function, where power (x, y) calculates x^y. Here is an implementation of power in an

ML-like language:

$$\begin{array}{lll} \textbf{fun} & \mathsf{power}\ (x, 0) & = 1 \\ \mid & \mathsf{power}\ (x, n) & = x * (\mathsf{power}\ (x, n-1)) \end{array}$$

The function power takes two arguments. Specializing it with respect to the second parameter, e.g., to the value 3, could lead to a residual program of the form

$$\textbf{fun}\ \ \mathsf{power\text{-}d\text{-}3}\ x\ =\ x * x * x * 1$$

Because the recursion is controlled by the static parameter, partial evaluation unfolds it four times, yielding a residual program without any recursive calls. In this case, partial evaluation seems to pay off: a considerable amount of calculation could be precomputed. However, consider now the result of specializing with respect to the first argument, again to the value 3:

$$\begin{array}{lll} \textbf{fun} & \mathsf{power\text{-}3\text{-}d}\ 0 & = 1 \\ \mid & \mathsf{power\text{-}3\text{-}d}\ n & = 3 * (\mathsf{power\text{-}3\text{-}d}\ (n-1)) \end{array}$$

Without knowledge about the second argument to power, there is hardly any computation that can be carried out—inlining the value of the first argument is about all that can be done. Clearly, partial evaluation cannot be used indiscriminately: The cost of partial evaluation has to be weighted against the gains of specialization.

Correctness of Partial Evaluation

Intuitively it is quite clear what correctness of partial evaluation means: specializing a program to a part of the input and then executing the residual program over some remaining input should yield the same result as executing the original program over the complete input, provided that both cases terminate.

A formal definition of this correctness requirement can be given with respect to a programming-language semantics: We write $[\![\mathsf{prog}]\!]$ for the denotation of a program(text) prog. For partial evaluation we consider programs whose input can be divided into two parts, namely a static part s and a dynamic part d. Hence running prog on the total input is written as $[\![\mathsf{prog}]\!]\ \langle s, d\rangle$. Let prog_s denote the specialization of prog to the static input s. Then the correctness requirement for specialization can be expressed as

$$[\![\mathsf{prog}_s]\!]\ d\ =\ [\![\mathsf{prog}]\!]\ \langle s, d\rangle \qquad (*)$$

Say that the program pe performs partial evaluation, i.e., $[\![\mathsf{pe}]\!]\ \langle \mathsf{prog}, s\rangle = \mathsf{prog}_s$. Combining this with $(*)$ yields the so-called *mix equation*

$$[\![[\![\mathsf{pe}]\!]\ \langle \mathsf{prog}, s\rangle]\!]\ d = [\![\mathsf{prog}]\!]\ \langle s, d\rangle \qquad (**)$$

as requirement for the correctness of pe.

Notice that for an ML program prog, we can form a trivial specialization prog_s by instantiating prog with the static input and abstracting over the dynamic parameters (providing, in fact a proof of Kleene's S_n^m *theorem* [34]):

$$\textbf{fn}\ x \Rightarrow \mathsf{prog}\ (s, x)$$

The real challenge of partial evaluation is to find non-trivial specializations, for which the parts of the computation that depend only on the static input have been carried out.

The Futamura Projections

The mix equation $(**)$ gives an equational specification of the behavior of a partial evaluator. Using this specification, applications of partial evaluation can be explored. For example, reasoning with the mix equation and an equational specifications of interpreters, compilers, and compiler generators, yields the so-called *Futamura projections*, first observed by Futamura [22, 23]. The Futamura projections describe how a partial evaluator can be used to compile, to generate compilers, and to generate compiler generators.

An interpreter for a programming language $\mathcal{L}$ can be understood as a function taking two arguments: a source program to be interpreted and some input for this program. Let $[\![\cdot]\!]_{\mathcal{L}}$ denote a semantic function assigning meanings to programs written in $\mathcal{L}$. A defining equation for an interpreter interpreter is then

$$[\![\mathsf{interpreter}]\!]\ \langle \mathsf{source}, \mathit{input}\rangle = [\![\mathsf{source}]\!]_{\mathcal{L}}\ \mathit{input}$$

Using a partial evaluator to specialize the interpreter to some source input, we can thus conclude (using the mix equation $(**)$) that

$$[\![[\![\mathsf{pe}]\!]\ \langle \mathsf{interpreter}, \mathsf{source}\rangle]\!]\ \mathit{input} = [\![\mathsf{source}]\!]_{\mathcal{L}}\ \mathit{input}$$

Consider now what a compiler does: When compiling some source to a target, the following equation should hold:

$$[\![\mathsf{target}]\!]\ \mathit{input} = [\![\mathsf{source}]\!]_{\mathcal{L}}\ \mathit{input}$$

By equational reasoning and extensionality follows the *first Futamura projection*:

$$[\![\mathsf{pe}]\!]\ \langle \mathsf{interpreter}, \mathsf{source}\rangle = \mathsf{target}$$

Recall that the equational treatment of partial evaluation is only half the story, since it only captures the extensional behavior of the resulting residual program. However since the control flow of an interpreter usually depends on the program to be interpreted, one can expect that all computations dealing with destructing the source program are carried out during partial evaluation. Thus we indeed can regard $[\![\mathsf{pe}]\!]\ \langle \mathsf{interpreter}, \mathsf{source}\rangle$ as a compiled program—the partial evaluator acts as a compiler.

Using a similar line of reasoning, one arrives at the *second Futamura projection*

$$[\![\mathsf{pe}]\!]\ \langle \mathsf{pe}, \mathsf{interpreter}\rangle = \mathsf{compiler},$$

i.e., specializing a partial evaluator with respect to an interpreter for some language $\mathcal{L}$ yields a compiler for $\mathcal{L}$—the partial evaluator acts as a compiler-generator. The *third Futamura projection* shows that partial evaluation can act as a compiler-generator generator:

$$[\![\mathsf{pe}]\!]\ \langle \mathsf{pe}, \mathsf{pe}\rangle = \mathsf{compiler\text{-}generator}$$

The Futamura projections can be generalized to arbitrary programs using the concept of a *generating extension*: A program p' is a generating extension of the program p, if running p' on static input s yields a specialization of p with respect to the s (under the assumption that p' terminates on s). Obviously, a compiler acts as the generating extension of an interpreter. The generalized Futamura projections read:

$$\begin{array}{lcl} [\![\mathsf{pe}]\!]\ \langle \mathsf{program}, \mathsf{static\ input}\rangle & = & \mathsf{specialized\ program} \\ [\![\mathsf{pe}]\!]\ \langle \mathsf{pe}, \mathsf{program}\rangle & = & \mathsf{generating\ extension} \\ [\![\mathsf{pe}]\!]\ \langle \mathsf{pe}, \mathsf{pe}\rangle & = & \mathsf{generating\text{-}extension\text{–}generator} \end{array}$$

The quest for self applicability

Both the second and the third Futamura projection require an application of a partial evaluator to itself. The Futamura projections thus became a driving force for the development of self-applicable partial evaluators. Only in 1985—fourteen years after the Futamura projections had been formulated—the very first effective self-applicable partial evaluator, Mix [33], was implemented by Jones's group at DIKU in Copenhagen. Subsequently, a number of self-applicable partial evaluators have been implemented, e.g., Similix [10]; whether a given partial evaluator is self applicable has become a standard question when classifying partial-evaluation systems.

Online and offline partial evaluation

The enabling technique used by Jones et al. in implementing a self-applicable partial evaluator was the development of *offline* partial evaluation.

The first partial evaluators to be written were *online*. Online partial evaluation works by interpreting a program text over the static and dynamic input in a non-standard way: the partial evaluator always keeps track of which values are static and which are dynamic. A computation that only depends on static values is carried out while for all other computations code is generated (residualizing the involved static values into syntax).

Online partial evaluation offers the most finely grained distinction between static and dynamic values that is possible. This however comes at a cost. Since for any function taking n arguments, 2^n cases have to be considered (any argument may be static or dynamic), online partial evaluators are rather large,

consisting of numerous case distinctions, and slow. Further, self application for online partial evaluators is bound to fail in practice: The decision whether to carry out a computation or to generate residual code generally depends on the static input, which is not available during self-application. The specialized partial evaluator thus still bears this overhead of decision-making.

Jones et al. decided to take the decision of whether to carry out computation or to generate code *offline*. They staged partial evaluation into (1) a so-called binding-time analysis and (2) the specialization of a program annotated with binding-time information. Binding-time analysis yields a conservative approximation about which values are static and which are dynamic. This information is then used during specialization. Compared with online partial evaluation, some possibilities for precomputation may be lost, but the partial evaluator becomes smaller and faster, and self application can be achieved efficiently.

1.1.2 Partial Evaluation in Practice

Partial evaluation optimizes programs by carrying out static computations, i.e., precomputing program parts that depend only on static input. Whether the optimization is worthwhile depends on (1) how much static computation is detected by the partial evaluator, and (2) how often this static computation is repeated when executing the original program. Repetition may be due to repeated execution of the complete program with the same static input, or loops/recursion within the program. Loops/recursion, however, can also lead to problems in the form of unreasonably large results of partial evaluation or in the form of non-termination of the partial-evaluation process.

Pitfalls

Code explosion, i.e., unreasonably large results of specialization, can occur, for example, when partial evaluation unrolls loops or unfolds recursive calls too eagerly. For example, when specializing the power function to a small exponent, unfolding the recursion is desirable; that is not necessarily the case for a large exponent, as illustrated by Debray [20]. Non-termination of partial evaluation may occur if the control flow of the program to be specialized is partly dynamic: When encountering a conditional with a dynamic test, the partial evaluator might specialize all branches of the conditional, speculatively, and in doing so may loop.

The danger of code explosion and non-termination can be minimized or even avoided with more sophisticated partial-evaluation systems; as an alternative, one can modify programs according to the capabilities of the partial evaluator in question.

Binding-time improvements

Not only code explosion and non-termination can be avoided by modifying a program according to the capabilities of the partial evaluator to be used: Also

how much static computation is detected by a partial evaluator can be influenced significantly by the way a program is written. Meaning-preserving program modifications that make program more suitable to partial evaluation, essentially by increasing the amount of static computation that is detected, are called *binding-time improvements*.

What kinds of binding-time improvements are necessary depends on the partial evaluator that is used, leading to a certain trade-off: A simple partial-evaluation system requires the user to carry out more binding-time improvements. For a simple partial evaluator, such binding-time improvements can be carried out in a systematic way, because the behavior of the partial evaluator is sufficiently predictable. A sophisticated partial-evalution system, in contrast, handles more programs automatically, but is more unstable in the sense that its behavior is less predictable.

Applications

At the beginning of this section we gave conditions under which partial evaluation is likely to be worthwhile. These may seem somewhat limited at first—in which situations are parts of the input known in advance? Also the fact that the user may have to modify programs so as to make them amenable to partial evaluation could raise doubts about the applicability of partial evaluation. Yet, as we shall see in the following, there are many successful applications of partial evaluation.

In extreme cases, even specializing a program with respect to *no* input may yield an improvement. For example, programs in which a large number of macros have been used may contain precomputable parts introduced by macro expansion. Also programs that are the result of combining a number of modules—such as programs written from reusable components—may benefit from specialization.

Repeated static computation occurs, for example, in C programs when reading or writing a large amount of data line by line, using formatting functions such as `printf` and `scanf`. These functions are passed a *control string* which determines their behavior; for a high number of repetitions, specializing them with respect to the control string is worthwhile. A similar situation arises with a number of Unix utilities such as `grep` or `awk` that scan a file line by line.

The Unix utilities `grep` and `awk` in fact interpret small programming languages; also `printf` and `scanf` can be seen as interpreters of the control string. Specializing interpreters so as to achieve the effect of compilation or for compiler generation—recall the Futamura projections from Section 1.1.1—is one of the prime applications of partial evaluation. Because programs often contain loops or recursion and furthermore are usually executed several times, specializing an interpreter with respect to a program is likely to be worthwhile. Furthermore, experience shows that interpreters are well-suited for partial evaluation. The main interest lies not so much on interpreters for full-scale programming languages such as C or ML, for which already exist highly optimized compilers, but on small *domain-specific languages* (DSLs). For example, partial evaluation

has been successfully used for rapid prototyping of domain specific languages, giving rise to a design methodology for DSLs that uses partial evaluation in a crucial way [14].

Often, a denotational semantics of a programming language can be implemented straightforwardly as an interpreter. Through the first and second Futamura projection, partial evaluation thus offers the possibility to extract a compiler directly from the language definition—we speak of *semantics-directed* compilation [30]. An immediate benefit of semantics-directed compilation is that correctness of the compilation process with respect to the semantics is guaranteed.

Other situations in which static computation is carried out repeatedly because of loops/recursion within a program can often be found in scientific computing. For example, the Fast Fourier Transformation (FFT), which converts a function defined in terms of time to a function defined in terms of frequency, has been shown to be well-suited for partial evaluation [36]. The input to the FFT includes a one-dimensional array of complex numbers and the size of the array. In applications, the FFT algorithm is often repeatedly applied to arrays of the same size. Further, the FFT carries out a fair amount of computation that only depends on the array size. Hence, a considerable speedup can be achieved by specializing the FFT with respect to the array size. Similar situations where partial evaluation is worthwhile have been found for example in applications from numerical computing [8, 9, 24], ray tracing [2], and media processing [21].

1.1.3 Contributions

In the following, we list the contributions of this dissertation to the field of partial evaluation.

The second Futamura projection for type-directed partial evaluation

Whether a given partial evaluator is self applicable is, as mentioned in Section 1.1.1, a standard questions when classifying partial-evaluation systems. We answer this question for type-directed partial evaluation [17, 18] (TDPE), a relatively recent partial evaluation technique.

TDPE can be seen as the combination of two concepts. The first concept could be called *partial evaluation by normalization*: The basic observation is that an appropriate notion of normalization corresponds to partial evaluation. The second concept is called *normalization by evaluation* (NbE): The normal form of a term t is extracted from the meaning of t by first evaluating t and then applying an extraction function to the result. Traditional normalizers instead work by symbolic computation on a representation of t. NbE yields a very efficient normalization function, and is thus of interest to any area of computer science in which normalization plays a role. NbE has for example been examined in the context of proof theory [7, 6]. The proceedings of the first APPSEM workshop on NbE [19] provide an overview of various other treatments.

TDPE instantiates the first concept with NbE as normalization function. There exist several fundamental differences between TDPE and 'traditional' partial evaluators, which—like traditional normalizers—work by symbolic computation on a code representation. Even though a limited form of self-application for TDPE already had been achieved in the original article [17], it was not clear whether self-application in general—in particular the second Futamura projection—could be performed in the setting of TDPE. This dissertation answers the question in the affirmative. We show how TDPE can be made amenable to self-application, present a formulation of the second Futamura projection suitable for TDPE, and derive an ML implementation of the second Futamura projection for TDPE. We further demonstrate our technique with several examples, including compiler generation for Tiny, a prototypical imperative language.

Semantics-directed compilation for goal-directed evaluation

We present the first application of semantics-directed compilation to a programming-language paradigm called *goal-directed evaluation* (GDE): We start with an intuitive semantics of GDE and end with efficient code comparable to that produced by optimized compilers.

Goal-directed languages such as Icon and Snobol combine expressions that can yield multiple results through backtracking. An expression can yield several results, which are generated one by one. If an expression fails to produce a result, control is passed to a previous expression, prompting it for another result. In case of success, the original expression is retried. The presence of backtracking complicates both language semantics and implementation for goal-directed languages.

In the literature, semantics and implementation of goal-directed languages have only been treated separately; we link semantics and implementation via semantics-directed compilation using partial evaluation. We further (1) relate an intuitive semantics of a goal-directed language conceptually and formally with a more complicated semantics that is well-suited for code-generation via partial evaluation, and (2) extract templates from the specialized interpreters that are similar to those found in an optimizing compiler described in the literature. All in all, we describe a unified approach to goal-directed evaluation with partial evaluation as one of the central techniques.

Partial evaluation of pattern matching in strings, revisited

After a non-trivial binding-time improvement, a naive, essentially quadratic string matcher can be specialized into a linear one à la Knuth, Morris and Pratt by standard partial evaluation. Several examples of string matchers that specialize well have been published. We revisit the problem, extracting an approach that is applicable to pattern matching problems in general, and relating various published solutions with program transformation.

A string matcher that searches for occurrences of a pattern string in a data

string can be seen as an interpreter of the pattern string—consequently, partial evaluation might be applied successfully. However, a straightforward implementation of a string matcher does not specialize well under basic partial evaluation: Too little static computation is detected. Successful application of partial evaluation requires either a non-trivial binding-time improvement of the naive matcher, or a partial-evaluation system with capabilities that go beyond standard partial evaluation.

Since Consel and Danvy [12] presented the first string matcher that specializes well under partial evaluation, many more applications of partial evaluation to string matching and, more generally, pattern matching have been published. Because in each case the (non-trivial!) binding-time improvement is performed in a single step, it is neither obvious that the modification preserves semantics, nor clear how such binding-time improvements can be achieved in a systematic way. We present a step-by-step modification such that only the first step is problem specific, whereas the remaining steps are variations of standard binding-time improvements; we hope that this division illuminates how the approach can be applied to other pattern-matching problems and implementations. We further explore a number of different string matchers that specialize well under standard partial evaluation, amongst them several published versions, and relate them with program transformations. All in all, we provide a comprehensive account of the application of partial evaluation to string matching.

1.2 Automatic cost analysis

Cost analysis, i.e., the prediction of the amount of resources such as space or running time used by an algorithm or program, originated in the field of algorithmics. Knuth's first volume of *The Art of Computer Programming* [35], which can be seen as the starting point of modern algorithmics, presented algorithm design with a strong focus on the analysis of running time. In fact, algorithm design and analysis are inseparable:

- Algorithmic design requires analysis, since resource requirements are an indispensable yardstick for the quality of an algorithm.
- Analysis often is based on correctness arguments for an algorithm, which in turn are integral part of the algorithm design.

The dependency of cost analysis on correctness arguments—in the most basic case a *termination* argument—shows that full automation of cost analysis for general algorithms or programs cannot be achieved. Even if the analysis depends only to a small extent on correctness arguments, a formidable amount of mathematics may be necessary to derive cost bounds: The analysis of the running time of recursive programs, for example, involves solving recurrence equations [1], for which a wealth of mathematical methods [5, 35, 40, 46] has been developed. Attempts to automate cost analysis therefore must compromise between the degree of automation, the class of algorithms or programs to be

considered, and the nature of the bounds to be determined. In the following, after reviewing a few basic concepts of cost analysis, we look at various research directions in the field of automatic cost analysis, examining for each which kind of compromise has been made. We then describe our contribution and put it into context. We shall mainly focus on time bounds rather than bounds for space or other resources.

1.2.1 Basic concepts of cost analysis

Successful analysis usually yields a cost approximation in terms of input size, expressed for example as a function. It is common practice to abstract from an actual function of cost in terms of input size to the function's *rate of growth*, yielding an *asymptotic analysis*, first advocated by Aho, Hopcroft and Ullman [1] as a means to compare relative performance.

The analysis of an algorithm is subject to how cost is modeled and what kind of approximation should be achieved. Cost models often are derived from machine models (e.g., the random-access machine or the pointer machine) by understanding the algorithm as a program running on such a machine and measuring cost accordingly. Other cost models are of a more abstract nature, e.g., counting the number of comparisons for sorting algorithms, or the number of recursive calls for functional programs.

Having decided on a cost model, the analysis is further determined by which kind of approximation is chosen:

- *Worst-case* analysis yields an upper bound of the running time of an algorithm for any input of a given size.
- *Average-case* analysis yields the *expected* running time based on a probabilistic model of the distribution of input. The rationale is that worst-case situations may occur very seldomly, so worst-case analysis is frequently too pessimistic when compared to running times observed in practice.
- *Amortized* analysis yields bounds on sequences of operations rather than on individual operations. Especially for data structures such as sets or queues, information about the time taken by a sequence of information may be the most appropriate. At the same time, flexibility is gained for the design of such data structures: instead of striving for a low worst-case bound for every individual operation, costs can be redistributed between operations; often amortized solutions are simpler and faster than worst-case solutions.

1.2.2 Approaches to automated cost analysis

In the following we look at various approaches to automated cost analysis, highlighting which compromise between the degree of automation, the class of algorithms or programs to be considered and the nature of the bounds to be determined has been made.

Implicit computational complexity

Implicit computational complexity examines how to give language-based rather than machine-based characterizations of complexity classes. Basically, for a given complexity class such as the polytime functions a language is defined such that all programs written in the language are in this class.

The field of implicit computational complexity started with the first machine-independent characterization of a complexity class: Cobham [11] showed that a certain class of function definitions characterized exactly the class of functions computable in polynomial time. However, it is undecidable to test whether a given function definition is legal. Further work in implicit computational complexity (for example [4, 39, 27]—Hofmann's survey [28] provides a detailed overview) managed to give more tangible characterizations and extended the language with, e.g., higher-order functions and inductive data types.

Even with language extensions as mentioned above, the recursion-theoretic formalisms used in work on implicit computational complexity are rather far from programming languages used in practice. Jones [31] developed an analysis for a *standard* first-order language that allows the automatic detection of polytime programs; the class of programs recognized by the analysis has been shown to be of practical value in that it admits many natural formulations of efficient algorithms. The analysis is an extension of work on termination analysis [38].

Resource-bound certification

Asymptotic bounds, as established through implicit computational complexity, do not provide enough information for the kind of safety guarantee necessary in real-time environments or when running code from untrusted sources. *Resource-bound certification* aims to provide more precise bounds, compromising either on the level of automation or on the class of programs to be considered.

Hughes and Pareto [29] present a functional language—a first-order variant of ML with constructs for explicit storage management—with a type system such that well-typed programs run within stated space bounds. Necula and Lee [45] show that the framework of *proof-carrying code* [44] can be used to express and check runtime bounds. Within the framework of typed assembly language [43], Crary and Weirich [16] develop a type system capable of specifying and certifying both space and time bounds.

All mentioned systems check rather than infer bounds, i.e., the authors compromised mainly on the level of automation. Reistad and Gifford [47], in contrast, consider a restricted language, namely functional programs without general recursion: Programs are written using combinators such as map and fold. The effects associated with function types are cost expressions that may depend on the size of input arguments; type inference calculates such cost expressions for every function type in the program, thus establishing time bounds. These bounds for used for guiding the parallelization of programs rather than as safety certificates.

Extraction of cost functions

Another possible compromise, allowing the inference of cost bounds for a general language, is to lower the requirements regarding the nature of the inferred bounds: a *cost function* p^c is a program that calculates a worst-case bound for a program p as a function of the size of input to p. A cost function may not even terminate for every input—using it as a certificate, therefore, is rather pointless. Cost functions instead can be used to make estimates about the resource consumption of a program. Such estimates are useful, for example, within automatic program transformation systems (e.g., the CACHET system [41]): They can give an indication of whether a transformation actually produced a better (more efficient, less space consuming, etc.) program. Transforming and approximating the cost function itself may yield a closed form, i.e., a formulation without recursion (or loops), so that the bound can be read off easily. This latter step, however, is equivalent to solving recurrence equations, so there is an inherent limit to its automation.

Le Métayer's ACE system [37] for a subset of FP [3] is based on program transformation: Drawing from a library of over 1000 rules, many of them tailored to recognize patterns of recursion, it aims to transform the program into a recursion-free cost bound[1] and a measure function, i.e., a function mapping input to input size.

Rosendahl [48] presents a system based on abstract interpretation [15] and program transformation that infers cost functions for first-order recursive equations. Rather than attempting to find a measure function in a completely automatic way, the method requires some user interaction in finding an appropriate size measure for data. As in the ACE system, program transformation is used, but only for simplifying an already established cost function rather than deriving such a function in the first place. Liu and Gómez [42] propose a method based on Rosendahl's work in which they use advanced program-transformation techniques to make the cost function more accurate and more efficient.

1.2.3 Our contribution

For our work, we chose the following compromise between the class of programs, the degree of automation, and the nature of the bounds:

Class of programs We consider first-order functional programs with inductive data types.

Degree of automation We require the user to define a size measure on input data.

Nature of bounds We extract recurrence equations—our method does not make any attempt to solve or simplify them.

[1] The ACE system reports failure if it does not succeed to eliminate recursion for the cost bound. Reporting instead the latest result of the transformation, however, often would yield a usable cost function.

Our key technical observation is that cost recurrences can be systematically extracted from type derivations in Dependent ML [49](DML). DML is a conservative extension of ML with a limited form of dependent types. The design philosophy of DML is to use type-checking for the verification of non-trivial correctness properties of ML programs—every valid ML program is a valid DML program, because DML extends ML conservatively. For example, DML types can express that consing an element to a list of length n yields a list of length $n + 1$, or that a program for inserting an element into a balanced tree indeed returns a balanced tree. In effect, data is associated with size (and shape) information—a size measure is encoded. The measure describes an abstraction from data to data size, which can be used to extract a recurrence equation from a program.

The method is used as follows:

- Finding an appropriate size measure is left to the user; the high expressiveness of DML types allows the user to tailor size measures to each situation.
- Once a size-measure has been defined and encoded with DML types, extracting a recurrence from a program is a tedious but systematic process that can be automated.
- In general, solving recurrences automatically is not possible, and thus we leave it to the user. Unlike systems that extract cost functions [37, 48], we choose not to carry out any approximations while extracting a recurrence. This approach may lead to recurrences that contain logical formulas; in contrast to cost functions, such recurrences are not immediately executable, but often specify a much more accurate bound.

All in all, our approach to automated cost analysis attempts a clean separation between what can be successfully automated and what is better left to the user or specialized tools; both a back-end for approximating recurrence equations naively into cost functions and for solving recurrence equations with further user interaction could be used.

1.3 Outline of the dissertation

Chapters 2 to 5 of this dissertation contain articles published in conference proceedings and/or scientific journals. In two cases, the dissertation contains an extended version of the original article.

- Chapter 2: Bernd Grobauer and Zhe Yang. The second Futamura projection for type-directed partial evaluation. *Higher-Order and Symbolic Computation*, 14(2/3), 2001. A preliminary version appeared in Julia L. Lawall, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 34, No 11, pages 22–32, Boston, Massachusetts, November 2000. ACM Press.

- Chapter 3: Olivier Danvy, Bernd Grobauer, and Morten Rhiger. A unified approach to goal-directed evaluation. In *New Generation Computing*, 20(1), 2001. A preliminary version appeared in Talid Waha, editor, *Proceedings of the 2001 International Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG)*, number 2196 in Lecture Notes in Computer Science, Florence, Italy, September 2001. Springer-Verlag.

- Chapter 4: Bernd Grobauer and Julia Lawall. Partial evaluation of pattern matching in strings, revisited. Accepted for publication in the *Nordic Journal of Computing*. The version included in this dissertation has been extended with several appendices.

- Chapter 5: Bernd Grobauer. Cost recurrences for DML programs. In Xavier Leroy, editor, *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming*, Florence, Italy, September 2001. ACM Press.

  The version included in this dissertation has been extended significantly.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] Peter Holst Andersen. Partial evaluation applied to ray tracing. In W. Mackens and S. M. Rump, editors, *Software Engineering in Scientific Computing*, pages 78–85. Vieweg, 1996.

[3] J. W. Backus. Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[4] Stephen Bellantoni and Cook Stephen. New recursion-theoretic characterization of the polytime functions. *Computational Complexity*, (2):97–110, 1992.

[5] Jon L. Bentley, Dorothea Haken, and James B. Saxe. A general method for solving divide-and-conquer recurrences. *SIGACT News*, 13(3):36–44, 1980.

[6] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In Bernhard Möller and J.V. Tucker, editors, *Prospects for hardware foundations (NADA)*, number 1546 in Lecture Notes in Computer Science, pages 117–137. Springer-Verlag, 1998.

[7] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In Albert R. Meyer, editor, *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science*, pages 203–213, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[8] Andrew A. Berlin. Partial evaluation applied to numerical computation. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 139–150, Nice, France, June 1990. ACM Press.

[9] Andrew A. Berlin and Rajeev J. Surati. Partial evaluation for scientific computing: The Supercomputer Toolkit experience. In Peter Sestoft and Harald Søndergaard, editors, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*,

Technical Report 94/9, University of Melbourne, Australia, pages 133–141, Orlando, Florida, June 1994.

[10] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[11] Alan Cobham. The intrinsic computational difficulty of functions. In Yehoshua Bar-Hillel, editor, *Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science*, pages 24–30. North Holland, 1964.

[12] Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, January 1989.

[13] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Graham [25], pages 493–501.

[14] Charles Consel and Renaud Marlet. Architecturing software using a methodology for language development. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *Tenth International Symposium on Programming Language Implementation and Logic Programming (PLILP'98); held jointly with the 6th International Conference, ALP'98*, number 1490 in Lecture Notes in Computer Science, pages 170–194, Pisa, Italy, September 1998. Springer-Verlag.

[15] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Ravi Sethi, editor, *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, January 1977.

[16] Karl Crary and Stephanie Weirich. Resource bound certification. In Thomas Reps, editor, *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 184–198, Boston Massachusetts, January 2000. ACM Press.

[17] Olivier Danvy. Type-Directed Partial Evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

[18] Olivier Danvy. Type-directed partial evaluation. In Hatcliff et al. [26], pages 367–411.

[19] Olivier Danvy and Peter Dybjer, editors. *Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98,* (Gothenburg, Sweden, May 8–9, 1998), number NS-98-8 in Note Series, DAIMI, Department of Computer Science, University of Aarhus, May 1998. BRICS.

[20] Saumya Debray. Resource-bounded partial evaluation. In Charles Consel, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 179–192, Amsterdam, The Netherlands, June 1997. ACM Press.

[21] Scott Draves. Implementing bit-addressing with specialization. In Mads Tofte, editor, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, Amsterdam, The Netherlands, June 1997. ACM Press.

[22] Yoshihito Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprinted from Systems · Computers · Controls 2(5), 1971.

[23] Yoshihito Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.

[24] Robert Glück, Ryo Nakashige, and Robert Zöchling. Binding-time analysis applied to mathematical algorithms. In J. Doležal and J. Fidler, editors, *System Modelling and Optimization*, pages 137–146. Chapman & Hall, 1995.

[25] Susan L. Graham, editor. *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993. ACM Press.

[26] John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors. *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, Copenhagen, Denmark, July 1998. Springer-Verlag.

[27] Martin Hofmann. Linear types and non size-increasing polynomial time computation. In Giuseppe Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 464–473, Trento, Italy, 1999. IEEE Computer Society Press.

[28] Martin Hofmann. Programming languages capturing complexity classes. *SIGACT News*, 31(1):31–42, March 2000.

[29] John Hughes and Lars Pareto. Recursion and dynamic datastructures in bounded space: Towards embedded ML programming. In Peter Lee, editor, *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 70–81, Paris, France, September 1999. ACM Press.

[30] Neil D. Jones, editor. *Semantics-Directed Compiler Generation*, number 94 in Lecture Notes in Computer Science, Aarhus, Denmark, 1980. Springer-Verlag.

[31] Neil D. Jones. Program analysis for implicit computational complexity. In Olivier Danvy and Andrzej Filinski, editors, *Programs as Data Objects, Second Symposium, PADO 2001*, number 2053 in Lecture Notes in Computer Science, page 1, Aarhus, Denmark, May 2001. Springer-Verlag. Joint invited talk of PADO 2001 and ICC 2001.

[32] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[33] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, February 1989. DIKU Report 91/12.

[34] Stephen C. Kleene. *Introduction to Metamathematics*. D. van Nostrand, Princeton, New Jersey, 1952.

[35] Donald E. Knuth. *Fundamental Algorithms,*volume 1 of *The Art of Computer Programming*. Addison-Wesley, 1968.

[36] Julia L. Lawall. Faster fourier transforms via automatic program specialization. In Hatcliff et al. [26], pages 338–355.

[37] Daniel Le Métayer. ACE: An automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, April 1988.

[38] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In Hanne Riis Nielson, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 81–92, London, United Kingdom, January 2001. ACM Press.

[39] Daniel Leivant. Stratified functional programs and computational complexity. In Graham [25], pages 325–333.

[40] Chung L. Liu. *Introduction to Combinatorial Mathematics*. McGraw-Hill, 1968.

[41] Yanhong Annie Liu. Efficiency by incrementalization: an introduction. *Higher-Order and Symbolic Computation (HOSC)*, 13(4):289–313, December 2000.

[42] Yanhong Annie Liu and Gustavo Gómez. Automatic accurate time-bound analysis for high-level languages. In Frank Mueller and Azer Bestavros, editors, *Proceedings of the ACM SIGPLAN 1998 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, number 1474 in LNCS, pages 31–40, Montréal, Canada, June 1998. Springer-Verlag.

[43] Greg Morrisett, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions Programming Languages and Systems*, 21(3):528–569, May 1999.

[44] Georg Necula. Proof-carrying code. In Neil D. Jones, editor, *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997. ACM Press.

[45] Georg Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In Giovanni Vigna, editor, *Special Issue on Mobile Agent Security*, number 1419 in Lecture Notes in Computer Science, pages 61–91. Springer-Verlag, October 1997.

[46] Paul W. Purdom, Jr and Cynthia A. Brown. *The Analysis of Algorithms*. Holt, Rinehart, and Winston, 1985.

[47] Brian Reistad and David K. Gifford. Static dependent costs for estimating program execution time. In Carolyn L. Talcott, editor, *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, Orlando, Florida, June 94. ACM Press.

[48] Mads Rosendahl. Automatic complexity analysis. In Joseph E. Stoy, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture '89*, pages 144–156, London, September 1989. ACM Press.

[49] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In Alex Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, Texas, January 1999. ACM Press.

Chapter 2

The Second Futamura Projection for Type-Directed Partial Evaluation

Abstract

A generating extension of a program specializes the program with respect to part of the input. Applying a partial evaluator to the program trivially yields a generating extension, but specializing the partial evaluator with respect to the program often yields a more efficient one. This specialization can be carried out by the partial evaluator itself; in this case, the process is known as the second Futamura projection.

We derive an ML implementation of the second Futamura projection for Type-Directed Partial Evaluation (TDPE). Due to the differences between 'traditional', syntax-directed partial evaluation and TDPE, this derivation involves several conceptual and technical steps. These include a suitable formulation of the second Futamura projection and techniques for making TDPE amenable to self-application. In the context of the second Futamura projection, we also compare and relate TDPE with conventional offline partial evaluation.

We demonstrate our technique with several examples, including compiler generation for Tiny, a prototypical imperative language.

2.1 Introduction

2.1.1 Background

General notions Given a general program $p\colon \sigma_S \times \sigma_D \to \sigma_R$ and a fixed *static* input $s : \sigma_S$, partial evaluation (a.k.a. program specialization) yields a specialized program $p_s : \sigma_D \to \sigma_R$. When this specialized program p_s is applied to an arbitrary *dynamic* input $d : \sigma_D$, it produces the same result as the original program applied to the complete input (s, d), i.e., $[\![p_s]\!]d = [\![p]\!](s, d)$ (Here, $[\![\cdot]\!]$ maps a piece of program text to its denotation. In this article, metavariables in *slanted serif* font, such as p, s, and d stand for program terms. Meanwhile, variables in *italic* font, such as x and y, are normal variables in the subject program). Often, some computation in program p can be carried out independently of the dynamic input d, and hence the specialized program p_s is more efficient than the general program p. In general, specialization is carried out by performing the computation in the source program p that depends only on the static input s, and generating program code for the remaining computation (called residual code). A partial evaluator *PE* is a program that performs partial evaluation automatically, i.e., if *PE* terminates on p and s then

$$[\![PE]\!](p, s) = p_s$$

Often extra annotations are attached to p and s so as to pass additional information to the partial evaluator.

A program p' is a generating extension of the program p, if running p' on s yields a specialization of p with respect to the static input s (under the assumption that p' terminates on s). Because the program $\lambda s.PE(p, s)$ computes a specialized program p_s for any input s, it is a trivial *generating extension* of program p. To produce a more efficient generating extension, we can specialize *PE* with respect to p, viewing *PE* itself as a program and p as part of its input. In the case when the partial evaluator *PE* itself is written in its input language, i.e., if *PE* is *self-applicable*, this specialization can be achieved by *PE* itself. That is, we can generate an efficient generating extension of p as

$$[\![PE]\!](PE, p).$$

Self-application The above formulation was first given in 1971 by Futamura [17] in the context of compiler generation—the generating extension of an interpreter is a compiler—and is called the *second Futamura projection*. Turning it into practice, however, proved to be much more difficult than what its seeming simplicity suggests; it was not until 1985 that Jones's group implemented Mix [23], the very first effective self-applicable partial evaluator. They identified the reason for previous failures: The decision whether to carry out computation or to generate residual code generally depends on the static input s, which is not available during self-application; so the specialized partial evaluator still bears this overhead of decision-making. They solved the problem by taking

the decision *offline*, i.e., the source program p is pre-annotated with binding-time annotations that solely determine the decisions of the partial evaluator. In the simplest form, a binding time is either static, which indicates computation carried out at partial evaluation time (hence called static computation), or dynamic, which indicates code-generation for the specialized program.

Subsequently, a number of self-applicable partial evaluators have been implemented, e.g., Similix [3], but most of them are for untyped languages. For typed languages, the so-called *type specialization* problem arises [21]: Generating extensions produced using self application often retain a universal data type and the associated tagging/untagging operations as a source of overhead. The universal data type is necessary for representing static values in the partial evaluator, just as it is necesssary for representing values in a standard evaluator. This is unsurprising, because a partial evaluator acts as a standard evaluator when all input is static.

Partly because of this, in the 1990's, the practice shifted towards hand-written generating-extension generators [2, 20]; this is also known as the *cogen approach*. Conceptually, a generating-extension generator is a staged partial evaluator, just as a compiler is a staged interpreter. Ideally, producing a generating extension through self-application of the partial evaluation saves the extra effort in staging a partial evaluator, since it reuses both the technique and the correctness argument of the partial evaluator. In practice, however, it is often hard to make a partial evaluator (or a partial-evaluation technique, as in the case of this paper) self-applicable in the first place. In terms of the correctness argument, if the changes to the partial evaluator in making it self-applicable are minor and are easily proved to be meaning-preserving, then the correctness of a generating extension produced by self-application still follows immediately from that of the partial evaluator.

As we shall see in this article, the problem caused by using a universal data type can be avoided to a large extent, if we can avoid introducing an implicit interpreter in the first place. The second Futamura projection thus still remains a viable alternative to the hand-written approach, as well as a source of interesting problems and a benchmark for partial evaluators.

Type-directed partial evaluation In a suitable setting, partial evaluation can be carried out by normalization. Consider, for example, the pure simply typed λ-calculus, in which computation means β-reduction. Given two λ-terms $p\colon \tau_1 \to \tau_2$ and $s\colon \tau_1$, bringing the application ps into β-normal form specializes p with respect to s. For example, normalizing the application of the K-combinator $K = \lambda x.\lambda y.x$ to itself yields $\lambda y.\lambda x.\lambda y'.x$.

Type-directed partial evaluation (TDPE), due to Danvy [5], realizes the above idea using a technique that turned out to be Berger and Schwichtenberg's *Normalization by Evaluation* (NbE) [1, 8]. Roughly speaking, NbE works by extracting the normal form of a term from its meaning, where the extraction function is coded in the object language.

Example 2.1 *Let PL be a higher-order functional language in which we can define a type* `Exp` *of term representations. Consider the combinator* $\mathsf{K} = \lambda x.\lambda y.x$*—the term* KK *is of type* $\texttt{Exp} \to \texttt{Exp} \to \texttt{Exp} \to \texttt{Exp}$*. We want to extract a* β*-normal form from its meaning.*

Since $\texttt{Exp} \to \texttt{Exp} \to \texttt{Exp} \to \texttt{Exp}$ *is the type of a function that takes three arguments, one can infer that a* β*-normal form of* KK *must be of the form* $\underline{\lambda\, v_1}.\underline{\lambda\, v_2}.\underline{\lambda\, v_3}.\mathsf{t}$ *(we underline term representations to distinguish them from terms), for some term* $\mathsf{t} : \texttt{Exp}$*. Intuitively, the only natural way to generate the term* t *from the meaning of term* KK *is to apply it to the term representations* $\underline{v_1}$*,* $\underline{v_2}$ *and* $\underline{v_3}$*. The result of this application is* $\underline{v_2}$*. Thus, we can extract the normal form of* KK *as* $\underline{\lambda\, v_1}.\underline{\lambda\, v_2}.\underline{\lambda\, v_3}.\underline{v_2}$*.*

TDPE is different from a traditional, syntax-directed offline partial evaluator [22] in several respects:

Binding-Time Annotation In traditional partial evaluation, all subexpressions require binding-time annotations. It is unrealistic for the user to annotate the program fully by hand. Fortunately, these annotations are usually computed by an automatic binding-time analyzer, while the user only needs to provide binding-time annotations on input arguments. On the other hand, since the user does not have direct control over the binding-time annotations, he often needs to know how the binding-time analyzer works and to tune the program in order to ensure termination and a good specialization result.

In contrast, TDPE eliminates the need to annotate expression forms that correspond to function, product and sum type constructions. One only needs to give a binding-time classification for the base types appearing in the types of constants. Consequently, it is possible, and often practical, to annotate the program by hand.

Role of Types The extraction function is parameterized over the type of the term to be normalized, which makes TDPE "type-directed".

Efficiency A traditional partial evaluator works by symbolic computation on the source programs; it contains an evaluator to perform the static evaluation and code generation. TDPE reuses the underlying evaluator (interpreter or compiler) to perform these operations; when run on a highly optimized evaluation mechanism, TDPE acquires the efficiency for free—a feature shared with the cogen approach.

Flexibility Traditional partial evaluators need to handle all constructs used in a subject program, evaluating the static constructs and generating code for the dynamic ones. In contrast, TDPE uses the underlying evaluator for the static part. Therefore, all language constructs can be used in the static part of a subject program. However, we shall see that this flexibility is lost when self-applying TDPE.

These differences have contributed to the successful application of TDPE in various contexts, e.g., to perform semantics-based compilation [12]. An introductory account, as well as a survey of various treatments concerning NbE, can be found in Danvy's lecture notes [7].

2.1.2 Our work

The problem A natural question is whether one can perform self-application, in particular the second Futamura projection, in the setting of TDPE. It is difficult to see how this can be achieved, due to the drastic differences between TDPE and traditional partial evaluation.

- TDPE extracts the normal form of a term according to a type that can be assigned to the term. This type is supplied in some form of encoding as an argument to TDPE. We can use self-application to specialize TDPE with respect to a particular type; the result helps one to visualize a particular instance of TDPE. This form of self-application was carried out by Danvy in his original article on TDPE [5]. However, it does not correspond to the second Futamura projection, because no further specialization with respect to a particular subject program is carried out.

- The aforementioned form of self-application [5] was carried out in the untyped language Scheme. Whether self-application can be achieved in a language with Hindley-Milner type system such as ML [25] is not immediately clear: Whereas TDPE can be implemented in Scheme as a function that takes a type encoding as its first argument, this strategy is impossible in ML, because such a function would require a dependent type. Indeed, the ML implementation of TDPE uses the technique of type encodings [32]: For every type, a particular TDPE program is constructed. As a consquence, the TDPE algorithm to be specialized is not fixed.

- Following the second Futamura projection literally, one should specialize the source program of the partial evaluator. In TDPE, the static computations are carried out directly by the underlying evaluator, which thus becomes an integral part of the TDPE algorithm. The source code of this underlying evaluator might be written in an arbitrary language or even be unavailable. In this case, writing this evaluator from scratch by hand is an extensive task. It further defeats the main point of using TDPE: to reuse the underlying evaluator and to avoid unnecessary interpretive overhead.

TDPE also poses some technical problems for self-application. For example, TDPE treats monomorphically typed programs, but the standard call-by-value TDPE algorithm uses the polymorphically typed control operators shift and reset to perform let-insertion in a polymorphically typed evaluation context.

Our contribution This article addresses all the above issues. We show how to effectively carry out self-application for TDPE in a language with Hindley-

Milner type system. To generate efficient generating extensions, such as compilers, we reformulate the second Futamura projection in a way that is suitable for TDPE.

More technically, for the typed setting, we show how to use TDPE on the combinators that constitute the TDPE algorithm and consequently on the type-indexed TDPE itself, and how to slightly rewrite the TDPE algorithm, so that we only use the control operators at the unit and boolean types. As a full-fledged example, we derive a compiler for the Tiny language.

Since TDPE is both the tool and the subject program involved in self-application, we provide a somewhat detailed introduction to the principle and the implementation of TDPE in Section 2.2. Section 2.3 provides an abstract account of our approach to self-application for TDPE, and Section 2.4 details the development in the context of ML. Section 2.5 describes the derivation of the Tiny compiler. Based on our experiments, we give some benchmarks in Section 2.6. Section 2.7 concludes. The appendix provides an index of notation (Appendix 2.A) and gives further technical details in the generation of a Tiny compiler (Appendix 2.B). The complete source code of the development presented in this article is available online [18].

2.2 TDPE in a nutshell

In order to give some intuition, we first outline TDPE for an effect-free fragment of ML without recursion. Then we sketch the extensions and pragmatic issues of TDPE in a larger subset of ML, which is the setting we will work with in the later sections. Finally, to facilitate a precise formulation of self-application, we outline Filinski's formalization of TDPE.

2.2.1 Pure TDPE in ML

In this section, we illustrate TDPE for an effect-free fragment of ML without recursion, which we call *Pure TDPE*. For this fragment, the call-by-name and call-by-value semantics agree, which allows us to directly use Berger and Schwichtenberg's NbE for call-by-name λ-calculus as the core algorithm (recall that ML is a call-by-value functional language).

NbE works by extracting the normal form of a λ-term from its meaning, by regarding the term as a higher-order code-manipulation function. The extraction functions are type-indexed coercion functions coded in the object language. To carry out partial evaluation based on NbE, TDPE thus needs to prepare a code-manipulation version of the subject λ-term. Such a λ-term, in general, could contain constant functions that cannot be statically evaluated; these constants have to be replaced with code-manipulation functions.

Pure simply-typed λ-terms We first consider TDPE only for pure simply-typed λ-terms. We use the type `Exp` in Figure 2.1 on the facing page to represent code (as it is used in Example 2.1 on page 23). In the following we will write $\underline{v}$

for `VAR` v, $\underline{\lambda}\, v.t$ for `LAM (`v`, `t`)` and $s\,\underline{@}\,t$ for `APP (`s`, `t`)`; following the convention of the λ-calculus, we use $\underline{@}$ as a left-associative infix operator.

```
datatype Exp = VAR of string
             | LAM of string * Exp
             | APP of Exp * Exp
```

Figure 2.1: A data type for representing terms

Let us for now only consider ML functions that correspond to pure λ-terms with type τ of the form $\tau ::= \bullet \mid \tau_1 \to \tau_2$, where '$\bullet$' denotes a base type. ML polymorphism allows us to instantiate '$\bullet$' with `Exp` when coding such a λ-term in ML. So every λ-term of type τ gives rise to an ML value of type $\ulcorner\tau\urcorner = \tau[\bullet := \texttt{Exp}]$; that is, a value representing either code (when $\ulcorner\tau\urcorner = \texttt{Exp}$), or a code-manipulation function (at higher types).

Figure 2.2 shows the TDPE algorithm: For every type τ, we define inductively a pair of functions $\downarrow^\tau : \ulcorner\tau\urcorner \to \texttt{Exp}$ (reification) and $\uparrow_\tau : \texttt{Exp} \to \ulcorner\tau\urcorner$ (reflection). Reification is the function that extracts a normal form from the value of a code-manipulation function, using reflection as an auxiliary function. We explain reification and reflection through the following examples.

$$
\begin{array}{rcl}
\downarrow^{\bullet} e & = & e \\
\downarrow^{\tau_1 \to \tau_2} f & = & \underline{\lambda}\, x.{\downarrow^{\tau_2}}\,(f({\uparrow_{\tau_1}}\,x)) \qquad (x \text{ is fresh}) \\
 & & \\
\uparrow_{\bullet} e & = & e \\
\uparrow_{\tau_1 \to \tau_2} e & = & \lambda\, x.{\uparrow_{\tau_2}}\,(e\,\underline{@}\,({\downarrow^{\tau_1}}\,x))
\end{array}
$$

Figure 2.2: Reification and reflection

Example 2.2 *We revisit the normalization of* KK *(Example 2.1 on page 23). For the type* $\bullet \to \bullet \to \bullet \to \bullet$ *the equations given in Figure 2.2 define reification as*

$$\downarrow^{\bullet\to\bullet\to\bullet\to\bullet} e = \underline{\lambda}\, x.\underline{\lambda}\, y.\underline{\lambda}\, z.e\,\underline{x}\,\underline{y}\,\underline{z}.$$

For every argument of base type '$\bullet$'*, a lambda-abstraction with a fresh variable name is created. Given a function of type* $\texttt{Exp} \to \texttt{Exp} \to \texttt{Exp} \to \texttt{Exp}$ *(i.e., a code-manipulation function), a code representation of the body is then generated by applying this function to the code representations of the three bound variables. Evaluating* $\downarrow^{\bullet\to\bullet\to\bullet\to\bullet}$ (KK) *yields* $\underline{\lambda}\, x.\underline{\lambda}\, y.\underline{\lambda}\, z.\underline{y}$.

What happens if we want to extract the normal form of $t : \tau_1 \to \tau_2$ where τ_1 is not a base type? The meaning of t cannot be directly applied to the code representing a variable, since the types do not match: $\ulcorner\tau_1\urcorner \neq \texttt{Exp}$. This is where

the *reflection* function $\uparrow_\tau : \texttt{Exp} \to \overline{\tau}$ comes in; it converts a code representation into a code-generation function:

Example 2.3 *Consider* $\tau_1 = \bullet \to \bullet \to \bullet \to \bullet$*:*

$$\uparrow_{\bullet\to\bullet\to\bullet\to\bullet} e = \lambda x.\lambda y.\lambda z.e\,\underline{@}\,x\,\underline{@}\,y\,\underline{@}\,z$$

For any term representation e*,* $\uparrow_{\bullet\to\bullet\to\bullet\to\bullet} e$ *is a function that takes three term representations and constructs a representation of the application of* e *to these term representations. It is used, e.g., when reifying the term* $\lambda x.\lambda y.xyyy$ *with* $\downarrow^{(\bullet\to\bullet\to\bullet\to\bullet)\to\bullet\to\bullet}$.

Adding constants So far we have seen that we can normalize a pure simply-typed λ-term by (1) coding it in ML, interpreting all the base types as type `Exp`, so that its value is a code-manipulation function, and (2) applying reification at the appropriate type. Treating terms with constants follows the same steps, but the situation is slightly more complicated. Consider, for example, the ML expression $\lambda z.$`mult` 3.14 z of type `real` $\to$ `real`, where `mult` is a curried version of multiplication over reals. This function cannot be used as a code-manipulation function. The solution is to use a non-standard, code-generation version $\texttt{mult}_\texttt{r} : \texttt{Exp} \to \texttt{Exp} \to \texttt{Exp}$ of `mult`. We also *lift* the constant 3.14 into `Exp` using a lifting-function $\mathit{lift}_\texttt{real} : \texttt{real} \to \texttt{Exp}$. (This operation requires a straightforward extension of the data type `Exp` with an additional constructor `LIT_REAL`.) Reflection can then be used to construct a code-generation version $\texttt{mult}_\texttt{r}$ of `mult`:

Example 2.4 *A code-generation version* $\texttt{mult}_\texttt{r}\texttt{: Exp}\to\texttt{Exp}\to\texttt{Exp}$ *of* $\texttt{mult}\texttt{: real}\to\texttt{real}\to\texttt{real}$ *is given by*

$$\texttt{mult}_\texttt{r} = \uparrow_{\bullet\to\bullet\to\bullet} \underline{\text{“}\mathit{mult}\text{”}} = \lambda x.\lambda y.\underline{\text{“}\mathit{mult}\text{”}}\,\underline{@}\,x\,\underline{@}\,y,$$

where $\underline{\text{“}\mathit{mult}\text{”}}$ *(=* `VAR` *“mult”) is the code representation of a constant with name mult. Now applying the reification function* $\downarrow^{\bullet\to\bullet}$ *to the term*

$$\lambda z.(\texttt{mult}_\texttt{r}\ (\mathit{lift}_\texttt{real}\ 3.14)\ z)$$

evaluates to $\underline{\lambda x.\text{“}\mathit{mult}\text{”}}\,\underline{@}\,\underline{3.14}\,\underline{@}\,\underline{x}$.

Partial evaluation

In the framework of TDPE, the partial evaluation of a (curried) program $p : \sigma_S \to \sigma_D \to \sigma_R$ with respect to a static input $s : \sigma_S$ is carried out by normalizing the application ps. We could use a code-generation version for all the constants in this term; reifying the meaning will carry out all the β-reductions, but leave all the constants in the residual program—no static computation involving constants is carried out. However, this is not good enough: One would expect that the application ps enables also computation involving constants, not only β-reductions. Partial evaluation, of course, should also carry out such

computation. This is achieved by instantiating the constants in question to themselves.

In general, to perform TDPE for a term, one needs to decide for each constant occurrence, whether to use the original constant or a code-generation instantiation of it; appropriate lifting functions have to be inserted where necessary. The result must type-check, and its partial application to the static input must represent a code-manipulation function (i.e., its type is built up from only the base type `Exp`), so that we can apply the reification function.

This process of classification corresponds to a binding-time annotation phase, as will be made precise in the framework of a two-level language (Section 2.2.3). Basically, a source term is turned into a well-formed two-level term by marking constants as static or dynamic, inserting lifting functions where needed. In general, one tries to reduce the number of occurrences of dynamic constants in term t, so that more static computation involving constants is carried out during partial evaluation. Because only constant occurrences have to be annotated, this can, in practice, be done by hand. Given an annotated term t_{ann}, we call the corresponding code-manipulation function its *residualizing instantiation* $\overline{t_{\mathsf{ann}}}$. It arises from t_{ann} by instantiating each dynamic constant c with its code-generation version c_{r}, each static constant with itself, and each lifting function with the appropriate coercion function into `Exp`. If t is of type σ, then its normal form can be calculated by reifying $\overline{t_{\mathsf{ann}}}$ at type σ (remember that reification only distinguishes a type's form—all base types are treated equally as '$\bullet$'):

$$NF(t) = [\![\downarrow^{\sigma} \overline{t_{\mathsf{ann}}}]\!];$$

Partial evaluation of a program $p : \sigma_S \times \sigma_D \to \sigma_R$ with respect to a static input $s : \sigma_S$ thus proceeds as follows:

- binding-time annotate p and s as p_{ann} and s_{ann}, respectively[1] such that the term $\lambda x.\overline{p_{\mathsf{ann}}}(\overline{s_{\mathsf{ann}}}, x)$ has type $\overline{\sigma_D \to \sigma_R}$ (recall that $\overline{\tau}$ arises from τ by instantiating each base type with `Exp`).
- carry out partial evaluation by reifying the above term at type $\sigma_D \to \sigma_R$:

$$p_s = [\![\downarrow^{\sigma_D \to \sigma_R} \lambda x.\overline{p_{\mathsf{ann}}}(\overline{s_{\mathsf{ann}}}, x)]\!]$$

Example 2.5 *Consider the function*

$$\texttt{height} = \lambda\,(a : \texttt{real}).\lambda\,(z : \texttt{real}).\texttt{mult}\ (\texttt{sin}\ a)\ z.$$

Suppose we want to specialize `height` *to a static input* a : `real`*. It is easy to see that the computation of* `sin` *can be carried out statically, but the computation of* `mult` *cannot—*`mult` *is a dynamic constant. This analysis results in a two-level term* $\texttt{height}_{\mathsf{ann}}$*, in which* `sin` *is marked as static,* `mult` *as dynamic, and a*

[1] That the static input also needs to be binding-time annotated may at first seem strange. This is natural, however, because TDPE also accepts higher-order values as static input. For a static input of base type, the binding-time annotation is trivial.

lifting function has been inserted to make the static result of applying `sin` *to a dynamic. The residualizing instantiation of* $\texttt{height}_{\mathsf{ann}}$ *instantiates* `sin` *with the standard sine function, the lifting function with a coercion function from* `real` *into* `Exp`*, and* `mult` *with a code-generation version as introduced in Example 2.4 on page 28:*

$$\overline{\texttt{height}_{\mathsf{ann}}} = \lambda\,(a : \texttt{real}).\lambda\,(z : \texttt{Exp}).\texttt{mult}_{\mathsf{r}}\ (\mathit{lift}_{\texttt{real}}(\texttt{sin}\ a))\ z$$

Now $(\overline{\texttt{height}_{\mathsf{ann}}}\ \frac{\pi}{6})$ *has type* $\texttt{Exp} \to \texttt{Exp}$*, i.e., it is a code-manipulation function. Thus, we can specialize* `height` *with respect to* $\frac{\pi}{6}$ *by evaluating* $\downarrow^{\bullet\to\bullet}(\overline{\texttt{height}_{\mathsf{ann}}}\ \frac{\pi}{6})$*, which yields* $\underline{\lambda\,x.\text{“}\underline{\mathit{mult}}\text{”}\,@\,\underline{0.5}\,@\,\underline{x}}$

Notice that instantiation in a binding-time annotated term t_{ann} of every constant with itself and of every lifting function with the identity function yields a term $\widetilde{t_{\mathsf{ann}}}$ that has the same denotation as the original term t; we call $\widetilde{t_{\mathsf{ann}}}$ the *evaluating instantiation* of t_{ann}.

2.2.2 TDPE in ML: implementation and extensions

Implementation Type-indexed functions such as reification and reflection can be implemented in ML employing a technique first used by Filinski and Yang [6, 32]; see also Rhiger's derivation [28]. A combinator is defined for every type constructor $\mathcal{T}$ ($\bullet$ and $\to$ in the case of Pure NbE in Section 2.2). This combinator takes a *pair* of reification and reflection functions for every argument τ_i to the (n-ary) type constructor $\mathcal{T}$, and computes the reification-reflection pair for the constructed type $\mathcal{T}(\tau_1, \ldots, \tau_n)$. Reification and reflection functions for a certain type τ can then be created by combining the combinators according to the structure of τ and projecting out either the reification or the reflection function.

As Figure 2.3 on the facing page shows, we specify these combinators in a signature called `NBE`. Their implementation as the functor `makePureNbE`—parameterized over two structures of respective signatures `EXP` (term representation) and `GENSYM` (name generation for variables)—is given in Figure 2.4 on page 32. The implementation is a direct transcription from the formulation in Section 2.2.1.

Example 2.6 *We implement an* `NBE`*-structure* `PureNbE` *by applying the functor* `makePureNbE` *(Figure 2.4 on page 32); this provides us with combinators* `-->` *and* `a'` *and functions* `reify` *and* `reflect`*. Normalization of* KK *(see Example 2.1 on page 23 and Example 2.2 on page 27) is carried out as follows:*

```
local  open PureNbE; infixr 5 -->  in
  val K = (fn x => fn y => x)
  val KK_norm = reify (a' --> a' --> a' --> a') (K K)
end
```

After evaluation, the variable `KK_norm` *is bound to a term representation of the normal form of* KK*.*

```
signature NBE =                                   (* normalization by evaluation *)
  sig
    type Exp
    type 'a rr                                    (* (↓^τ, ↑_τ) : τ rr *)

    val a' : Exp rr                               (* τ = • *)
    val --> : 'a rr * 'b rr -> ('a -> 'b) rr      (* τ = τ1 → τ2 *)
      ⋮
    val reify: 'a rr -> 'a -> Exp                 (* ↓^τ *)
    val reflect: 'a rr -> Exp -> 'a               (* ↑_τ *)
  end

signature EXP =                                   (* term representation *)
  sig
    type Exp
    type Var

    val VAR: Var -> Exp
    val LAM: Var * Exp -> Exp
    val APP: Exp * Exp -> Exp
     ⋮
  end

signature GENSYM =                                (* name generation *)
  sig
    type Var
    val new: unit -> Var                          (* make a new name *)
    val init: unit -> unit                        (* reset name counter *)
  end;
```

Figure 2.3: NbE in ML, signatures

Encoding two-level terms through functors

As mentioned earlier, the input to TDPE needs to be binding-time annotated, i.e., the input is a two-level term. The ML module system makes it possible to encode a two-level term p in a convenient way: Define p inside a functor `p_pe(structure D: DYNAMIC) = ...` which parameterizes over all dynamic types, dynamic constants and lifting functions. By instantiating `D` with an appropriate structure, one can create either the evaluating instantiation $\widetilde{p}$ or the residualizing instantiation $\overline{p}$.

Example 2.7 *In Example 2.5 on page 29 we sketched how the function* `height` *can be partially evaluated with respect to its first argument. Figure 2.5 on page 33 shows how to provide both evaluating and residualizing instantiation in ML using functors. The two-level term* $\texttt{height}_{\texttt{ann}}$ *is encoded as a functor*

```
functor makePureNbE(structure G: GENSYM
                    structure E: EXP
                    sharing type E.Var = G.Var): NBE =
  struct
    type Exp = E.Exp
    datatype 'a rr = RR of ('a -> Exp) * (Exp -> 'a)
                                                  (* (↓^τ, ↑_τ) : τ rr *)

    infixr 5 -->
    val a' = RR(fn e => e, fn e => e)                        (* τ = • *)
    fun RR (reif1, refl1) --> RR(reif2, refl2)          (* τ = τ1 → τ2 *)
      = RR (fn f =>
              let val x = G.new ()
                in E.LAM (x, reif2 (f (refl1 (E.VAR x))))
              end,
            fn e =>
              fn v => refl2 (E.APP (e, reif1 v)))

    fun reify (RR (reif, refl)) v                            (* ↓^τ *)
      = (G.init (); reif v)
    fun reflect (RR (reif, refl)) e                          (* ↑_τ *)
      = refl e
  end
```

Figure 2.4: Pure NbE in ML, implementation

`height_pe(structure D:DYNAMIC)` *that is parameterized over the dynamic type* `Real`*, the dynamic constant* `mult`*, and the lifting function* `lift_real` *in* $\texttt{height}_{\texttt{ann}}$.

Extensions We will use a much extended version of TDPE, referred to as *Full TDPE* in this article. Full TDPE not only treats the function type constructor, but also tuples and sums. Furthermore, a complication that we have disregarded so far is that ML is a call-by-value language with computational effects. In such languages, the β-rule is not sound because it might discard or duplicate computations with effects.

Extending TDPE to tuples is straightforward: reifying a tuple is done by producing the code of a tuple constructor and applying it to the reified components of the tuple; reflection at a tuple type means producing code for a projection on every component, reflecting these code pieces at the corresponding component type and tupling the results.

Sum types and call-by-value languages can be handled by manipulating the code-generation context in the reflection function. This has been achieved by using the control operators shift and reset [9, 14]. Section 2.4.5 describes in more detail the treatment of sum types and call-by-value languages in TDPE.

Figure 2.6 on page 34 displays the signature `CTRL` of control operators and

```
signature DYNAMIC =          (* Signature of dynamic types and constants *)
  sig
    type Real

    val mult: Real -> Real -> Real
    val lift_real: real -> Real
  end

                                   (* The functor encodes a two-level term *)
functor height_pe(structure D: DYNAMIC) =
 struct
   fun height a z = D.mult (D.lift_real (sin a)) z
 end


structure EDynamic: DYNAMIC =                              (* Defining ~· *)
  struct
    type Real = real
    fun mult x y  =  x * y
    fun lift_real r = r
  end

structure RDynamic: DYNAMIC =                              (* Defining ‾· *)
  struct
    local
      open EExp PureNbE
      infixr 5 -->
    in
      type Real = Exp
      val mult = reflect (a' --> a' --> a') (VAR "mult")
      fun lift_real r  = LIT_REAL r
    end
  end


structure Eheight = height_pe (structure D = EDynamic);
                                                    (* height~ann *)
structure Rheight = height_pe (structure D = RDynamic);
                                                   (* ‾height_ann‾ *)
```

Figure 2.5: Instantiation via functors

```
signature CTRL =                                        (* control operators *)
  sig
    type Exp
    val shift: (('a -> Exp) -> Exp) -> 'a
    val reset: (unit -> Exp) -> Exp
  end;

functor makeFullNbE(structure G: GENSYM
                    structure E: EXP
                    structure C: CTRL
                    sharing ... ): NBE = ...
```

Signatures `GENSYM`, `EXP`, and `NBE` are defined in Figure 2.3 on page 31.

Figure 2.6: Full NbE in ML.

the skeleton of a functor `makeFullNbE`, which implements Full TDPE—an implementation can be found in Danvy's lecture notes [7]. The relevance of Full TDPE in this article is that (1) it is the partial evaluator that one would use for specializing realistic programs; and (2) in particular, it handles all features used in its own implementation, including side effects and control effects. Hence in principle self-application should be possible.

2.2.3 A general account of TDPE

The introduction to TDPE given in Section 2.2.1 is concerned with providing intuition rather than formal detail; in the following, we describe Filinski's formalization of TDPE [16], which gives a precise definition to the concepts that were introduced only informally before. This formal account is rather technical and may be skipped on first reading: When developing self-application for TDPE in Section 2.3, we shall start with an intuitive account that can be understood without having read the following material. Nevertheless, the details of the development turn out to be rather intricate, so an informal account alone is not satisfactory. In Section 2.3.2 we draw upon the formal account of TDPE presented here, and derive a formulation of self-application from it.

Preliminaries First we fix some standard notions. A simple functional language is given by a pair $(\Sigma, \mathcal{I})$ of a signature Σ and an interpretation $\mathcal{I}$ of this signature. More specifically, the syntax of valid terms and types in this language is determined by Σ, which consists of base type names, and constants with types constructed from the base type names. (The types are possibly polymorphic; however, in our technical development, we will only work with monomorphic instances.) A set of typing rules generates, from the signature Σ, typing judgments of the form $\Gamma \vdash_\Sigma t : \sigma$, which reads "$t$ is a well-formed term of type σ under typing context Γ".

The denotational semantics of types and terms is determined by an interpretation. An interpretation $\mathcal{I}$ of signature Σ assigns domains to base type names, elements of appropriate domains to literals and constants, and, in the setting of call-by-value languages with effects, also monads to various effects. The interpretation $\mathcal{I}$ extends canonically to the meaning $[\![\sigma]\!]^{\mathcal{I}}$ of every type σ and the meaning $[\![t]\!]^{\mathcal{I}}$ of every term $t : \sigma$ in the language; we write $[\![t]\!]^{\mathcal{I}}$ for closed terms t, which denote elements in the domain $[\![\sigma]\!]^{\mathcal{I}}$.

The syntactic counterpart of the notion of an interpretation is that of an instantiation, which compositionally maps syntactic phrases in a language L to syntactic phrases in (usually) another language L'. The following definition of instantiations uses the notion of substitution. For a substitution Φ, we write $t\{\Phi\}$ and $\sigma\{\Phi\}$ to denote the application of Φ to term t and type σ, respectively.

Definition 2.8 (Instantiation) *Let L and L' be two languages with signatures Σ and Σ', respectively. An instantiation Φ of Σ-phrases (terms and types) into language L' is a substitution that maps the base types in Σ to Σ'-types, and maps constants $c : \sigma$ to closed Σ'-terms of type $\sigma\{\Phi\}$.*

We also refer to the term $t\{\Phi\}$ as the instantiation of the term t under Φ, and the type $\sigma\{\Phi\}$ as the instantiation of the type σ under Φ.

It should be obvious that an interpretation of a language L' and an instantiation of a language L in language L' together determine an interpretation of L.

Two-level language Filinski formalized TDPE using a notion of two-level languages (or, binding-time-separated languages). The signature $\Sigma^{\mathfrak{2}}$ of such a language is the disjoint union of a static signature $\Sigma^{\mathfrak{s}}$ (static base types $\mathsf{b}^{\mathfrak{s}}$ and static constants $c^{\mathfrak{s}}$, written with superscript $\mathfrak{s}$), a dynamic signature $\Sigma^{\mathfrak{d}}$ (dynamic base types $\mathsf{b}^{\mathfrak{d}}$ and dynamic constants $c^{\mathfrak{d}}$, written with superscript $\mathfrak{d}$), and lifting functions $\$_{\mathsf{b}}$ for base types. For simplicity, we assume all static base types $\mathsf{b}^{\mathfrak{s}}$ are persistent (a.k.a. liftable), i.e., each of them has a corresponding dynamic base type $\mathsf{b}^{\mathfrak{d}}$, and is equipped with a lifting function $\$_{\mathsf{b}} : \mathsf{b}^{\mathfrak{s}} \to \mathsf{b}^{\mathfrak{d}}$. The intuition is that a value of a persistent base type always has a unique external representation as a constant, which can appear in the generated code; we call such a constant a *literal*. The meaning $[\![e]\!]^{\mathcal{I}^{\mathfrak{2}}}$ of a term e is determined by a static interpretation $\mathcal{I}^{\mathfrak{s}}$ of signature $\Sigma^{\mathfrak{s}}$, and a dynamic interpretation $\mathcal{I}^{\mathfrak{d}}$ of signature $\Sigma^{\mathfrak{d}}$ and the lifting functions; we also write $[\![e]\!]^{\mathcal{I}^{\mathfrak{s}},\mathcal{I}^{\mathfrak{d}}}$ for $[\![e]\!]^{\mathcal{I}^{\mathfrak{2}}}$. A two-level language is different from a one-level language in that the meaning of terms is parameterized over the dynamic interpretation $\mathcal{I}^{\mathfrak{d}}$. More precisely, it is specified by a pair $(\Sigma^{\mathfrak{2}}, \mathcal{I}^{\mathfrak{s}})$ of its signature $\Sigma^{\mathfrak{2}}$ and a fixed static interpretation $\mathcal{I}^{\mathfrak{s}}$.

A two-level language $PL^{\mathfrak{2}} = (\Sigma^{\mathfrak{2}}, \mathcal{I}^{\mathfrak{s}})$ is usually associated with a one-level language $PL = (\Sigma^{PL}, \mathcal{I}^{PL})$:

1. The dynamic signature $\Sigma^{\mathfrak{d}}$ of $PL^{\mathfrak{2}}$ duplicates Σ^{PL} (except for literals, which can be lifted from static literals) with all constructs superscripted by $\mathfrak{d}$.

2. The static signature $\Sigma^{\mathfrak{s}}$ of PL^2 comprises all the base types in PL and all the constants in PL that have no computational effects except possible divergence. All these constructs are superscripted by $\mathfrak{s}$ in $\Sigma^{\mathfrak{s}}$.

3. The static interpretation $\mathcal{I}^{\mathfrak{s}}$ is the restriction of interpretation $\mathcal{I}^{PL}$ to $\Sigma^{\mathfrak{s}}$.

For clarity, we let metavariable t range over one-level terms, e over two-level terms, σ over one-level types, and τ over two-level types.

We can induce an *evaluating dynamic interpretation* $\mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}$ from $\mathcal{I}^{PL}$ by taking $[\![\mathsf{b}^{\mathfrak{d}}]\!]^{\mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![\mathsf{b}]\!]^{\mathcal{I}^{PL}}$, $[\![c^{\mathfrak{d}}]\!]^{\mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![c]\!]^{\mathcal{I}^{PL}}$, and $[\![\$_{\mathsf{b}}]\!]^{\mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = (\lambda x.x) \in [\![\mathsf{b}]\!]^{\mathcal{I}^{PL}} \to [\![\mathsf{b}]\!]^{\mathcal{I}^{PL}}$. A closely related notion is the *evaluating instantiation* of Σ^2-phrases in Σ^{PL}:

Definition 2.9 (Evaluating Instantiation) *The* evaluating instantiation *of a Σ^2-term $\vdash_{\Sigma^2} e : \tau$ in PL is $\vdash_{\Sigma^{PL}} \widetilde{e} : \widetilde{\tau}$, given by $\widetilde{e} = e\{\Phi_\sim\}$ and $\widetilde{\tau} = \tau\{\Phi_\sim\}$, where instantiation $\Phi_\sim$ is a substitution of Σ^2-constructs (constants and base types) into Σ^{PL}-phrases (terms and types): $\Phi_\sim(\mathsf{b}^{\mathfrak{s}}) = \Phi_\sim(\mathsf{b}^{\mathfrak{d}}) = \mathsf{b}$, $\Phi_\sim(c^{\mathfrak{s}}) = \Phi_\sim(c^{\mathfrak{d}}) = c$, $\Phi_\sim(\$_{\mathsf{b}}) = \lambda x.x$.*

We have that for all Σ^2-types τ and Σ^2-terms e, $[\![\widetilde{\tau}]\!]^{\mathcal{I}^{PL}} = [\![\tau]\!]^{\mathcal{I}^{\mathfrak{s}}, \mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}}$ and $[\![\widetilde{e}]\!]^{\mathcal{I}^{PL}} = [\![e]\!]^{\mathcal{I}^{\mathfrak{s}}, \mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}}$.

Static normalization Static normalization works on Σ^2-terms of fully dynamic type, i.e., having a type constructed solely from dynamic base types. A term is in *static normal form* if it is free of β-redexes and free of static constants, except literals that appear as arguments to lifting functions; in other words, the term cannot be further simplified without knowing the interpretations of the dynamic constants. Terms e in static normal form are, in fact, in one-to-one correspondence with terms $\widetilde{e}$ in Σ^{PL}. They can thus be represented using a one-level term representation such as the one provided by `Exp`.

A static normalization function NF for PL^2 is a computable partial function on well-typed Σ^2-terms such that if $e' = NF(e)$ then e' is a Σ^2-term in static normal form, and e and e' are not distinguished by any dynamic interpretation $\mathcal{I}^{\mathfrak{d}}$ of $\Sigma^{\mathfrak{d}}$, i.e., $\forall \mathcal{I}^{\mathfrak{d}}.[\![e]\!]^{\mathcal{I}^{\mathfrak{s}}, \mathcal{I}^{\mathfrak{d}}} = [\![e']\!]^{\mathcal{I}^{\mathfrak{s}}, \mathcal{I}^{\mathfrak{d}}}$; in other words, term e' and term e have the same (parameterized) meaning. Notice that NF is usually partial, since terms for which the static computation diverges have no normal form.

Normalization by evaluation In this framework, NbE can be described as a technique to reduce the static normalization function NF for a two-level language PL^2 to evaluation in the ordinary language PL. For this to be possible, we assume that language PL is equipped with a base type `Exp` for the representation of its own terms (and thus of static normal forms in PL^2), and constants that support name generation and code construction (for example, a lifting function $lift_{\mathsf{b}} : \mathsf{b} \to \mathtt{Exp}$ for every base type b).

Filinski has shown that in the described setting, NbE can be performed with two type-indexed functions $\downarrow^{\tau} : \overline{\tau} \to \mathtt{Exp}$ (reification) and $\uparrow_{\tau} : \mathtt{Exp} \to \overline{\tau}$ (reflection)—here the operation $\overline{\cdot}$ on two-level types corresponds to the one introduced in Section 2.2.1 for ML types; a formal definition is given below

in Definition 2.10. The function $\downarrow^\tau$ extracts the static normal form of a term $\vdash_{\Sigma^2} e{:}\tau$ from a special *residualizing instantiation* of the term in PL, $\vdash_{\Sigma^{PL}} \overline{e}{:}\overline{\tau}$, and the function $\uparrow_\tau$ is used in both the definition of reification function and the construction of the residualizing instantiation $\overline{e}$.

Definition 2.10 (Residualizing Instantiation) *The* residualizing instantiation *of a Σ^2-term $\vdash_{\Sigma^2} e : \tau$ in PL is $\vdash_{\Sigma^{PL}} \overline{e} : \overline{\tau}$, given by $\overline{e} = e\{\Phi_{\sqcap}\}$ and $\overline{\tau} = \tau\{\Phi_{\sqcap}\}$, where instantiation $\Phi_{\sqcap}$ is a substitution of Σ^2-constructs into Σ^{PL}-phrases: for base types b, $\Phi_{\sqcap}(\mathsf{b}^{\mathsf{s}}) = \mathsf{b}$, $\Phi_{\sqcap}(\mathsf{b}^{\mathsf{d}}) = \mathtt{Exp}$, for constants c, $\Phi_{\sqcap}(c^{\mathsf{s}}) = c$, $\Phi_{\sqcap}(c^{\mathsf{d}} : \tau) = \uparrow_\tau$ "$\underline{c}$", and for lifting functions over a base type b, $\Phi_{\sqcap}(\$_{\mathsf{b}}) = \mathit{lift}_{\mathsf{b}}$.*

In words, the residualizing instantiation $\overline{\tau}$ of a fully dynamic type τ substitutes the type `Exp` for all occurrences of dynamic base types in τ. Since type τ is fully dynamic, type $\overline{\tau}$ is constructed from type `Exp`, and thus represents a code value or a code manipulation function (see Section 2.2.1). The residualizing instantiation $\overline{e}$ of a term e substitutes all the occurrences of dynamic constants and lifting functions with the corresponding code-generation versions.[2]

The function NF in NbE is defined by Equation (2.1) in Figure 2.7 on the next page: It computes the static normal form of term e by evaluating the Σ^{PL}-term $\vdash_{\Sigma^{PL}} \downarrow^\tau \overline{e}{:}\mathtt{Exp}$ using an evaluator for language PL. In Filinski's semantic framework for TDPE, a correctness theorem of NbE has the following form, though the exact definition of function NF varies depending on the setting.

Theorem 2.11 (Filinski [16]) *The function NF defined in Equation (2.1) in Figure 2.7 on the following page is a static normalization function. That is, for all well-typed Σ^2-terms e, if $e' = NF(e)$, then term e' is in static normal form, and $\forall \mathcal{I}^{\mathsf{d}}.[\![e]\!]^{\mathcal{I}^{\mathsf{s}},\mathcal{I}^{\mathsf{d}}} = [\![e']\!]^{\mathcal{I}^{\mathsf{s}},\mathcal{I}^{\mathsf{d}}}$.*

Just as self-application reduces the technique of producing an efficient generating extension to the technique of partial evaluation, our results on the correctness of self-application reduce to Theorem 2.11. The details of how Theorem 2.11 is proved are out of the scope of this article.

Partial evaluation Given a Σ^2-term $\vdash_{\Sigma^2} p{:}\tau_S \times \tau_D \to \tau_R$, and its static input $\vdash_{\Sigma^2} s{:}\tau_S$, where both type τ_D and type τ_R are fully dynamic, specialization can be achieved by applying NbE (Equation (2.1) on the next page) to statically normalize the trivial specialization $\lambda x.p(s, x)$:

$$\begin{array}{rcl} NF(\lambda x.p(s,x)) & = & [\![\downarrow^{\tau_D\to\tau_R} \overline{\lambda x.p(s,x)}\,]\!]^{\mathcal{I}^{PL}} \\ & = & [\![\downarrow^{\tau_D\to\tau_R} \lambda x.\overline{p}(\overline{s}, x)]\!]^{\mathcal{I}^{PL}} \end{array} \tag{2.2}$$

In the practice of partial evaluation, one usually is not given two-level terms to start with. Instead, we want to specialize ordinary programs. This can be

[2] Compare with Example 2.5 on page 29, where $\mathtt{height}_{\mathsf{ann}}$ is

$$\lambda\,(a : \mathtt{real}^{\mathsf{s}}).\lambda\,(z : \mathtt{real}^{\mathsf{d}}).\mathtt{mult}^{\mathsf{d}}\ (\$_{\mathtt{real}}(\mathtt{sin}^{\mathsf{s}}\ a))\ z.$$

Normalization by Evaluation
For term $\vdash_{\Sigma^2} e : \tau$, we use

$$NF(e) = [\![\downarrow^{\tau} \overline{e}]\!]^{\mathcal{I}^{PL}} \tag{2.1}$$

to compute its static normal form, where
1. Term $\vdash_{\Sigma^{PL}} \overline{e} : \overline{\tau}$ is the residualizing instantiation of term e, and
2. Term $\vdash_{\Sigma^{PL}} \downarrow^{\tau} : \overline{\tau} \to \mathtt{Exp}$ is the reification function for type τ.

Binding-time annotation The task is, given $\vdash_{\Sigma^{PL}} t : \sigma$ and binding-time constraints in the form of a two-level type τ whose erasure is σ, to find $\vdash_{\Sigma^2} t_{\mathsf{ann}} : \tau$ that satisfies the constraints and the following equations:

$$[\![\tau]\!]^{\mathcal{I}^{\mathfrak{s}}, \mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![\sigma]\!]^{\mathcal{I}^{PL}} \qquad [\![t_{\mathsf{ann}}]\!]^{\mathcal{I}^{\mathfrak{s}}, \mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![t]\!]^{\mathcal{I}^{PL}}$$

Figure 2.7: A formal recipe for NbE

reduced to the specialization of two-level terms through a binding-time annotation step. For TDPE, the task of binding-time annotating a Σ^{PL}-term t with respect to some knowledge about the binding-time information of the input is, in general, to find a two-level term t_{ann} such that (1) the evaluating instantiation $[\![t_{\mathsf{ann}}]\!]^{\mathcal{I}^{\mathfrak{s}}, \mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}}$ of term t_{ann} agrees with the meaning $[\![t]\!]^{\mathcal{I}^{PL}}$ of term t, and (2) term t_{ann} is compatible with the input's binding-time information in the following sense: Forming the application of t_{ann} to the static input results in a term of fully dynamic type. Consequently, the resulting term can be normalized with the static normalization function NF.

Consider again the standard form of partial evaluation. We are given a Σ^{PL}-term $\vdash_{\Sigma^{PL}} p{:}\sigma_S \times \sigma_D \to \sigma_R$ and the binding-time information of its static input s of type σ_S, but not the static input s itself. The binding-time information can be specified as a Σ^2-type τ_S such that $\widetilde{\tau_S} = \sigma_S$; for the more familiar first-order case, type σ_S is some base type b, and type τ_S is simply $\mathsf{b}^{\mathfrak{s}}$. We need to find a two-level term $\vdash_{\Sigma^2} p_{\mathsf{ann}} : \tau_S \times \tau_D \to \tau_R$, such that (1) types τ_D and τ_R are the fully dynamic versions of types σ_D and σ_R, and (2) $[\![p_{\mathsf{ann}}]\!]^{\mathcal{I}^{\mathfrak{s}}, \mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![p]\!]^{\mathcal{I}^{PL}}$.

For a given static input $s : \sigma_S$, we want to normalize term $t \equiv \lambda x.p(s, x)$. Given a properly annotated $s_{\mathsf{ann}} : \tau_S$ ("properly" in the sense that $[\![s_{\mathsf{ann}}]\!]^{\mathcal{I}^{\mathfrak{s}}, \mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![s]\!]^{\mathcal{I}^{PL}}$), we can form the two-level term $\vdash_{\Sigma^2} t_{\mathsf{ann}} \equiv \lambda x.p_{\mathsf{ann}}(s_{\mathsf{ann}}, x) : \tau_D \to \tau_R$. By compositionality of the meaning functions, $[\![t_{\mathsf{ann}}]\!]^{\mathcal{I}^{\mathfrak{s}}, \mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![t]\!]^{\mathcal{I}^{PL}}$. If the term $e = NF(t_{\mathsf{ann}})$ is the result of the NbE algorithm, we see that its one-level representation $\widetilde{e}$, which we regard as the result of the specialization, has the same meaning as the term t:

$$[\![\widetilde{e}]\!]^{\mathcal{I}^{PL}} = [\![e]\!]^{\mathcal{I}^{\mathfrak{s}}, \mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![t_{\mathsf{ann}}]\!]^{\mathcal{I}^{\mathfrak{s}}, \mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![t]\!]^{\mathcal{I}^{PL}}$$

This verifies the correctness of the specialization.

Our setting In this article, the language *PL* we will work with is essentially ML, with a base type `Exp` for encoding term representations, the constructors associated with `Exp`, constants for name generations (`GENSYM.init` and `GENSYM.new`), and control operators. All of these can be introduced into ML as user-defined data types and functions; in practice, we do not distinguish between *PL* and ML. The associated two-level language PL^2 is constructed from the language *PL* mechanically. As shown in Section 2.2.2 (Example 2.7 on page 31), a two-level term can be encoded in ML by using a functor to parameterize over all dynamic types and constants in the term. Instantiating the functor with a structure that defines either the original constants or their code-generation versions yields the evaluating instantiation or the residualizing instantiation, respectively.

2.3 Formulating self-application

In this section, we present two forms of self-application for TDPE. One uses self-application to generate more efficient reification and reflection functions for a type τ; following Danvy [5], we refer to this form of self-application as *visualization*. The other adapts the second Futamura projection to the setting of TDPE. We first give an intuitive account of how self-application can be achieved, and then derive a precise formulation of self-application, based on the formal account of TDPE presented in Section 2.2.3.

2.3.1 An intuitive account of self-application

We start by presenting the intuition behind the two forms of self application, drawing upon the informal account of TDPE in Section 2.2.1.

Visualization

For a specific type τ, the reification function $\downarrow^\tau$ contains one β-redex for each recursive call following the type structure. For example, the direct unfolding of $\downarrow^{\bullet\to\bullet\to\bullet\to\bullet}$, according to its definition (Figure 2.2 on page 27), is

$$\lambda f_0.\underline{\lambda x}.(\lambda f_1.\underline{\lambda y}.(\lambda f_2.\underline{\lambda z}.(\lambda e.e)(f_2((\lambda e.e)\underline{z})))(f_1((\lambda e.e)\underline{y})))(f_0((\lambda e.e)\underline{x}))$$

rather than the normalized form presented in Example 2.2 on page 27. Normalization of such a function can be achieved by self-applying TDPE so as to specialize the reification function with respect to a particular type. Danvy has carried out this form of self application in the untyped language Scheme [5]; in the following, we reconstruct it in our setting.

Recall from Section 2.2 that finding the normal form of a term $t : \sigma$ is achieved by reifying the residualizing instantiation of a binding-time annotated version of t:

$$NF(t) = [\![\downarrow^\sigma \overline{t_{\mathsf{ann}}}]\!].$$

It thus suffices to find an appropriate binding-time annotated version of the term $\downarrow^\tau$. A straightforward analysis of the implementation of NbE (see Figure 2.3 on page 31 and Figure 2.4 on page 32), shows that all the base types (`Exp`, `Var`, etc.) and constants (`APP`, `Gensym.init`, etc.[3]) are needed in the code generation phase; hence they all should be classified as dynamic. Therefore, to normalize $\downarrow^\tau : \overline{\tau} \to \texttt{Exp}$, we use a trivial binding-time annotation, notated as $\underline{\langle\ .\ \rangle}$, in which every constant is marked as dynamic:

$$NF(\underline{\langle \downarrow^\tau \rangle}) = [\![\downarrow^{\tau \to \bullet}\ \overline{\underline{\langle \downarrow^\tau \rangle}}]\!], \tag{2.3}$$

In order to understand the term $\overline{\underline{\langle \downarrow^\tau \rangle}}$, we analyze the composite effect of the residualizing instantiation and trivial binding-time annotation: for a term e, the term $\overline{\underline{\langle e \rangle}}$ is formed from e by substituting all constants with their code-generation counterparts. We write $\Downarrow^\tau$ for $\overline{\underline{\langle \downarrow^\tau \rangle}}$ and $\Uparrow_\tau$ for $\overline{\underline{\langle \uparrow_\tau \rangle}}$ for notational conciseness.

Term $\downarrow^\tau$ and term $\Downarrow^\tau$ are respectively the evaluating instantiation and residualizing instantiation of the same (two-level) term $\underline{\langle \downarrow^\tau \rangle}$: that is, $\widetilde{\underline{\langle \downarrow^\tau \rangle}} = \downarrow^\tau$, and $\overline{\underline{\langle \downarrow^\tau \rangle}} = \Downarrow^\tau$; term $\uparrow_\tau$ and term $\Uparrow_\tau$ have an analogous relationship. We will exploit this fact in Section 2.4.1 to apply the functor-based approach to the reification/reflection combinators themselves, thus providing an implementation of $\Downarrow^\tau$ and $\Uparrow_\tau$ in ML.

Adapted second Futamura projection

As we have argued in the introduction, in the setting of TDPE, following the second Futamura projection literally is not a reasonable choice for deriving efficient generating extensions. The evaluator for the language in which we use TDPE might not even be written in this language. Furthermore, making an evaluator explicit in the partial evaluator to be specialized introduces an extra layer of interpretation, which defeats the advantages of TDPE. We thus consider instead the general idea behind the second Futamura projection:

> *Using partial evaluation to perform the static computations in a 'trivial' generating extension (usually) yields a more efficient generating extension.*

Following the informal recipe for performing TDPE given in Section 2.2, the 'trivial generating extension' $p^\dagger$ of a program $p : \sigma_S \times \sigma_D \to \sigma_R$ is

$$\lambda s.\, TDPE(p, s) : \sigma_S \to \texttt{Exp} = \lambda\, s.{\downarrow^{\sigma_D \to \sigma_R}}\ \lambda\, d.\overline{p_{\mathsf{ann}}}(s, d)$$

Since the trivial generating extension is itself a term, we can normalize it using TDPE: We reify at type $\sigma_S \to \bullet$ the residualizing instantiation of the (suitably binding-time annotated) trivial generating extension. We can use the trivial binding-time annotation, i.e., to reify $\overline{\underline{\langle\ \lambda s.\, TDPE(p, s)\ \rangle}}$—in Section 2.3.2 we shall explain in detail why this choice is actually not too conservative. Because

[3] These constants appear, e.g., in the underlined portion of the expanded term $\downarrow^{\bullet \to \bullet \to \bullet \to \bullet}$.

$\overline{\langle\,\underline{\cdot}\,\rangle}$ is a substitution, it distributes over term constructors, and we can move it inside the terms:

$$\overline{\langle\,\underline{\lambda s.\,TDPE(\mathsf{p},s)}\,\rangle} = \lambda s.\,\Downarrow^{\sigma_D\to\sigma_R}\,(\lambda d.\,\overline{\langle\,\underline{\mathsf{p}_{\mathsf{ann}}}\,\rangle}\,(s,d)).$$

For concreteness, the reader might find it helpful to consider the example of the height function (Example 2.5 on page 29): $\overline{\mathsf{p}_{\mathsf{ann}}}$ corresponds to $\overline{\mathtt{height}_{\mathsf{ann}}}$, so $\overline{\langle\,\underline{\mathsf{p}_{\mathsf{ann}}}\,\rangle}$ is formed by substituting all the constants in $\overline{\mathtt{height}_{\mathsf{ann}}}$ with their code-generation versions. Such constants include `sin`, $\mathit{lift}_{\mathtt{real}}$, and the code-constructing constants appearing in term $\mathtt{mult}_{\mathsf{r}}$ (Example 2.4 on page 28).

In practice, however, we do not need to first build the residualizing version by hand and then apply the TDPE formulation. Instead, we show that we can characterize $\overline{\langle\,\underline{\mathsf{e}}\,\rangle}$ in terms of the original two-level term e itself, thus enabling a functor-based approach: We write $\overline{\mathsf{e}}$ for $\overline{\langle\,\underline{\mathsf{e}}\,\rangle}$ and call it the *GE-instantiation* of term e, where "GE" stands for *generating extension*. A precise definition of the GE-instantiation is derived formally in Section 2.3.2 (Definition 2.17 on page 44). Basically, $\overline{\mathsf{e}}$ instantiates all static constants and lifting functions in e with their code-generation version and all dynamic constants with versions that generate "code-generation" code. In other words, static constants and lifting functions give rise to code that is executed when applying the generating extension, whereas dynamic constants give rise to code that has to appear in the result of applying the generating extension.

All in all, the generating extension $\mathsf{p}^{\ddagger}$ of a program $\mathsf{p} : \sigma_S \times \sigma_D \to \sigma_R$ can be calculated as

$$\mathsf{p}^{\ddagger} = [\![\downarrow^{\sigma_S\to\bullet}\,(\lambda s.\,\Downarrow^{\sigma_D\to\sigma_R}\,(\lambda d.\overline{\mathsf{p}_{\mathsf{ann}}}\,(s,d)))]\!]. \tag{2.4}$$

2.3.2 A derivation of self-application

In Section 2.3.1 we gave an intuitive account of how self-application can be achieved for TDPE. Using the formalization of TDPE presented in Section 2.2.3 we now derive both forms of self-application; correctness thus follows from the correctness of TDPE.

Visualization

We formally derive visualization (Section 2.3.1), using the "recipe" outlined in Figure 2.7 on page 38. First, we need a formal definition of the trivial binding-time annotation $\langle\,\underline{\cdot}\,\rangle$ in terms of the two-level language:

Definition 2.12 (Trivial Binding-Time Annotation) *The* trivial binding-time annotation *of a Σ^{PL}-term $\vdash_{\Sigma^{PL}} \mathsf{t}{:}\sigma$ is a PL^2-term $\vdash_{\Sigma^2} \langle\,\underline{\mathsf{t}}\,\rangle{:}\langle\,\underline{\sigma}\,\rangle$, given by $\langle\,\underline{\mathsf{t}}\,\rangle = \mathsf{t}\{\Phi_{\langle\,\rangle}\}$ and $\langle\,\underline{\sigma}\,\rangle = \sigma\{\Phi_{\langle\,\rangle}\}$, where the instantiation $\Phi_{\langle\,\rangle}$ is a substitution of Σ^{PL}-constructs into Σ^2-phrases: $\Phi_{\langle\,\rangle}(\mathsf{b}) = \mathsf{b}^{\mathfrak{d}}$, $\Phi_{\langle\,\rangle}(\ell : \mathsf{b}) = \$_{\mathsf{b}}\ell^{\mathfrak{s}}$ (ℓ is a literal), $\Phi_{\langle\,\rangle}(c) = c^{\mathfrak{d}}$ (c is not a literal).*

Lemma 2.13 (Properties of $\langle\underline{\cdot}\rangle$) *For a Σ^{PL}-term $\vdash_{\Sigma^{PL}} t : \sigma$, the following properties hold:*

1. $[\![\langle\underline{t}\rangle]\!]^{\mathcal{I}^s, \mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![t]\!]^{\mathcal{I}^{PL}}$, *making* $\langle\underline{t}\rangle$ *a binding-time annotation of* t;
2. $\widetilde{\langle\underline{t}\rangle} = t$;
3. $\langle\underline{\sigma}\rangle$ *is always a fully dynamic type;*
4. *If a Σ^2-type τ is fully dynamic, then* $\overline{\langle\underline{\overline{\tau}}\rangle} = \overline{\tau}$.

A simple derivation using properties (3) and (4) in Lemma 2.13, together with the fact that $\langle\underline{\cdot}\rangle$ and $\overline{\cdot}$ distribute over all type and term constructors, yields the formulation of self-application given in Equation (2.3) on page 40:

$$NF(\langle\underline{\downarrow^\tau}\rangle) = [\![\downarrow^{\tau\to\bullet}(\Downarrow^\tau)]\!]^{\mathcal{I}^{PL}}.$$

The following corollary follows immediately from Theorem 2.11 on page 37 and property (1) of Lemma 2.13.

Corollary 2.14 *If $e_\tau = NF(\langle\underline{\downarrow^\tau}\rangle)$, then its one-level representation $\widetilde{e_\tau}$ is free of β-redexes and is semantically equivalent to $\downarrow^\tau$:*

$$[\![\widetilde{e_\tau}]\!]^{\mathcal{I}^{PL}} = [\![e_\tau]\!]^{\mathcal{I}^s, \mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![\langle\underline{\downarrow^\tau}\rangle]\!]^{\mathcal{I}^s, \mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![\downarrow^\tau]\!]^{\mathcal{I}^{PL}}$$

The self-application carried out by Danvy in the setting of Scheme [5] is quite similar; his treatment explicitly λ-abstracts over the constants occurring in $\downarrow^\tau$, which, by the TDPE algorithm, would be reflected according to their types. This reflection also appears in our formulation: For any constant $c : \sigma$ appearing in $\downarrow^\tau$, we have $\overline{\langle\underline{c}\rangle} = \overline{c^{\mathfrak{d}}} = \uparrow_{\langle\underline{\sigma}\rangle} \text{“}\underline{c}\text{”}$. Consequently, our result coincides with Danvy's.

Adapted second Futamura projection

We repeat the development from Section 2.3.1 in a formal way. We begin by rederiving the trivial generating extension, this time from Equation (2.2) on page 37: In order to specialize a two-level term $\vdash_{\Sigma^2} p : \tau_S \times \tau_D \to \tau_R$ with respect to a static input $\vdash_{\Sigma^2} s : \tau_S$, we execute the Σ^{PL}-program $\vdash_{\Sigma^{PL}} \downarrow^{\tau_D\to\tau_R} \lambda d.\overline{p}(\overline{s}, d) : \mathtt{Exp}$. By λ-abstracting over the residualizing instantiation $\overline{s}$ of the static input s, we can trivially obtain a generating extension $p^\dagger$, which we will refer to as the trivial generating extension.

$$\vdash_{\Sigma^{PL}} p^\dagger \equiv \lambda s.\downarrow^{\tau_D\to\tau_R}(\lambda d.(\overline{p}(s,d))) : \overline{\tau_S} \to \mathtt{Exp}.$$

Corollary 2.15 (Trivial Generating Extension) *The term $p^\dagger$ is a generating extension of program p.*

Since the term $p^\dagger$ is itself a Σ^{PL}-term, we can follow the recipe in Figure 2.7 on page 38 to specialize it into a more efficient generating extension. We first need to binding-time annotate the term $p^\dagger$. For the subterm $\downarrow^{\tau_D \to \tau_R}$, the analysis in Section 2.3.1 shows that we should take the trivial binding-time annotation. For the subterm $\overline{p}$, the following analysis shows that it is not too conservative to take the trivial binding-time annotation as well. Since $\overline{\cdot} = \Phi_{\lceil\cdot\rceil}$ is an instantiation, i.e., a substitution on dynamic constants and lifting functions, every constant c' in $\overline{p}$ must appear as a subterm of the image of a constant or a lifting function under the substitution $\Phi_{\lceil\cdot\rceil}$. If c' appears inside $\Phi_{\lceil\cdot\rceil}(c^{\mathfrak{d}}) = \uparrow_\tau \underline{\text{“}c\text{”}}$ (where c' could be a code-constructor such as `LAM`, `APP` appearing in term $\uparrow_\tau$) , or $\Phi_{\lceil\cdot\rceil}(\$_{\mathsf{b}}) = \mathit{lift}_{\mathsf{b}}$, then c' is needed in the code generation phase, and hence it should be classified as dynamic. If c' appears inside $\Phi_{\lceil\cdot\rceil}(c^{\mathfrak{s}}) = c$, then $c' = c$ is an original constant, classified as static assuming the input s is given. Such a constant could rarely be classified as static in $p^\dagger$, since the input s is not statically available at this stage.

Taking the trivial binding time annotation of the trivial generating extension $p^\dagger$, we then proceed with Equation (2.1) on page 38 to generate a more efficient generating extension.

$$\begin{aligned}
p^\ddagger &= NF(\underline{\langle\, \lambda s.\ \downarrow^{\tau_D \to \tau_R}\ (\lambda d.(\overline{p}\ (s,d)))\,\rangle}) \\
&= [\![\downarrow^{\underline{\langle \tau_S \to \bullet \rangle}}\ \overline{\underline{\langle\, \lambda s.\ \downarrow^{\tau_D \to \tau_R}\ (\lambda d.(\overline{p}\ (s,d)))\,\rangle}}\ ]\!]^{\mathcal{I}^{PL}} \\
&= [\![\downarrow^{\tau_S \to \bullet}\ (\lambda s.\ \overline{\underline{\langle \downarrow^{\tau_D \to \tau_R} \rangle}}\ (\lambda d.(\ \overline{\underline{\langle \overline{p} \rangle}}\ \ (s,d))))]\!]^{\mathcal{I}^{PL}}
\end{aligned}$$

Expressing $\overline{\underline{\langle \overline{p} \rangle}}$ as $\overline{p}$, and $\overline{\underline{\langle \downarrow^{\tau_D \to \tau_R} \rangle}}$ as $\Downarrow^{\tau_D \to \tau_R}$, we have

$$p^\ddagger = [\![\downarrow^{\tau_S \to \bullet}\ (\lambda s.\ \Downarrow^{\tau_D \to \tau_R}\ (\lambda d.\overline{p}\ (s,d)))]\!]^{\mathcal{I}^{PL}},$$

as originally given in Equation (2.4) on page 41.

The generation of $p^\ddagger$ always terminates, even though, in general, the normalization function NF may diverge. Recall that the trivial binding-time annotation used in the preceding computation of $p^\ddagger$ marks all constants, including all fixed-point operators, as dynamic. Divergence can only happen when the two-level program contains static fixed-point operators.

Correctness of the second Futamura projection follows from Corollary 2.15 on the facing page and Theorem 2.11 on page 37.

Corollary 2.16 (Efficient Generating Extension) *Program $\widetilde{p^\ddagger}$ (that is, the one-level form of the static normal form $p^\ddagger$) is a generating extension of p which is free of β-redexes.*

Proof: By Theorem 2.11 on page 37 and the property of trivial binding-time analysis, we have $p^\ddagger$ is in static normal form, and $[\![\widetilde{p^\ddagger}]\!]^{\mathcal{I}^{PL}} = [\![p^\dagger]\!]^{\mathcal{I}^{PL}}$. That the program $\widetilde{p^\ddagger}$ is a generating extension of p follows from Corollary 2.15 on page 42.

□

Now let us examine how the term $\ulcorner \overline{p} \urcorner$ is formed. Note that $\ulcorner \overline{p} \urcorner = \overline{\langle \ulcorner p \urcorner \rangle} = ((p\{\Phi_{\ulcorner\urcorner}\})\{\Phi_{\langle\rangle}\})\{\Phi_{\ulcorner\urcorner}\} = p\{\Phi_{\ulcorner\urcorner} \circ \Phi_{\langle\rangle} \circ \Phi_{\ulcorner\urcorner}\}$; thus $\overline{\ulcorner \cdot \urcorner}$ corresponds to the composition of three instantiations, $\Phi_{\text{gm}} = \Phi_{\ulcorner\urcorner} \circ \Phi_{\langle\rangle} \circ \Phi_{\ulcorner\urcorner}$, which is also an instantiation. We call Φ_{gm} the generating-extension instantiation (GE-instantiation); a simple calculation gives its definition.

Definition 2.17 (GE-instantiation) *The* GE-instantiation *of a* Σ^2*-term* $\vdash_{\Sigma^2} e : \tau$ *in PL is* $\vdash_{\Sigma^{PL}} \ulcorner \overline{e} \urcorner : \ulcorner \overline{\tau} \urcorner$ *given by* $\ulcorner \overline{e} \urcorner = e\{\Phi_{\text{gm}}\}$ *and* $\ulcorner \overline{\tau} \urcorner = \tau\{\Phi_{\text{gm}}\}$*, where instantiation* Φ_{gm} *is a substitution of* Σ^2*-constructs into* Σ^{PL}*-phrases:*

$$\begin{aligned}
\Phi_{\text{gm}}(\mathsf{b}^{\mathsf{s}}) &= \Phi_{\text{gm}}(\mathsf{b}^{\mathsf{d}}) = \texttt{Exp} \\
\Phi_{\text{gm}}(c^{\mathsf{s}} : \tau) &= \overline{\langle \underline{c : \ulcorner \tau \urcorner} \rangle} = \uparrow_{\langle \ulcorner \tau \urcorner \rangle} \underline{\text{“}c\text{”}} \\
\Phi_{\text{gm}}(c^{\mathsf{d}} : \tau) &= \overline{\uparrow_\tau \underline{\text{“}c\text{”}}} = \Uparrow_\tau \overline{\langle \underline{\texttt{VAR}} \rangle} (\mathit{lift}_{\texttt{string}} \text{“}c\text{”}) \\
\Phi_{\text{gm}}(\$_b) &= \uparrow_{\bullet\to\bullet} \underline{\text{“}\mathit{lift}_b\text{”}}
\end{aligned}$$

Note that at some places, we intentionally keep the $\overline{\ulcorner \cdot \urcorner}$ form unexpanded, since we can just use the functor-based approach to obtain the residualizing instantiation. Indeed, the GE-instantiation boils down to “taking the residualizing instantiation of the residualizing instantiation”. In Section 2.4.3, we show how to extend the instantiation-through-functor approach to cover GE-instantiation as well.

It is instructive to compare the formulation of the second Futamura projection with the formulation of TDPE (Equation (2.2) on page 37). The crucial common feature is that the subject program p is only instantiated, i.e., only the constants are substituted in the program; this feature makes them amenable to a functor-based treatment and frees them from an explicit interpreter. For TDPE, however, static constants are instantiated with their standard instantiation, which makes it possible to use built-in constructs (such as case expressions) in the “static parts” of a program. This is not the case for the second Futamura projection, which causes some inconvenience when applying the second Futamura projection, as we shall see in Section 2.5.

2.4 The implementation

In this section we treat various issues arising when implementing the abstract formulation of Section 2.3 in ML. We start with the implementation of the key components for self application, namely the functions $\Downarrow$ and $\Uparrow$, and the GE-instantiation. We then turn to two technical issues. First, we show how to specify the input, especially the types, for the self-application. Second, we show how to modify the full TDPE algorithm, which uses polymorphically typed control operators, such that it is amenable to the TDPE algorithm itself, i.e., amenable to self-application.

2.4.1 Residualizing instantiation of the combinators

In Section 2.3.1 we remarked that the terms $\downarrow^\tau$ and $\Downarrow^\tau$ are respectively the evaluating instantiation and the residualizing instantiation of the same two-level term $\langle\underline{\downarrow^\tau}\rangle$. We can again use the ML module system to conveniently implement both instantiations. Recall that we formulated reification and reflection as type-indexed functions, and we implemented them not as a monolithic program, but as a group of combinators, one for each type constructor. These combinators can be plugged together following the structure of a type τ to construct a type encoding as a reification-reflection pair $(\downarrow^\tau, \uparrow_\tau)$. To binding-time annotate $(\downarrow^\tau, \uparrow_\tau)$ as $(\langle\underline{\downarrow^\tau}\rangle, \langle\underline{\uparrow_\tau}\rangle)$, it suffices to parameterize all the combinators over the constants they use: As already mentioned, because $\langle\underline{\ \cdot\ }\rangle$ is a substitution, it distributes over all constructs in a term, marking all the types and constants as dynamic. The combinators, when instantiated with either an evaluating or a residualizing instantiation, can be combined according to a type τ to yield either $(\downarrow^\tau, \uparrow_\tau)$ or $(\Downarrow^\tau, \Uparrow_\tau)$.

```
structure EExp                                  (* Evaluating Inst. ~· on EXP *)
= struct
    type Var = string
    datatype Exp =
        VAR of string                                              (* v *)
      | LAM of string * Exp                                        (* λx.e *)
      | APP of Exp * Exp                                           (* e1 @ e2 *)
      | PAIR of Exp * Exp                                          (* (e1,e2) *)
      | PFST of Exp                                                (* fst *)
      | PSND of Exp                                                (* snd *)
      | LIT_REAL of real                                           (* $real *)
  end

structure EGensym                          (* Evaluating Inst. ~· on GENSYM *)
= struct
    type Var = string

    local val n = ref 0
    in fun new () = (n := !n + 1;                    (* make a new name *)
                     "x" ^ Int.toString (!n))
       fun init () = n := 0                          (* reset name counter *)
    end
  end;

(* Evaluating Instantiation *)
structure EFullNbE = makeFullNbE (structure G = EGensym
                                  structure E = EExp
                                  structure C = ECtrl): NBE
```

Figure 2.8: Evaluating Instantiation of NbE

```
structure RExp: EXP = struct
  type Exp = EExp.Exp
  type Var = EExp.Exp

                                                    (* VAR v  =  VAR @ v *)
  fun VAR v = EExp.APP (EExp.VAR "VAR", v)

                                                (* LAM (v,e)  =  LAM @ (v,e) *)
  fun LAM (v, e) = EExp.APP (EExp.VAR "LAM",
                             EExp.PAIR (v, e))
                                                 (* APP (s,t)  =  APP @ (s,t) *)
  fun APP (s, t) = EExp.APP (EExp.VAR "APP",
                             EExp.PAIR (s, t))
  ⋮
end

⋮

                                              (* Residualizing Instantiations *)
structure RFullNbE = makeFullNbE (structure G = RGensym
                                  structure E = RExp
                                  structure C = RCtrl): NBE

structure RPureNbE = makePureNbE (structure G = RGensym
                                  structure E = RExp): NBE
```

Figure 2.9: Residualizing Instantiation of NbE

We can directly use the functors `makePureNbE` (Figure 2.4 on page 32) and `makeFullNbE` (Figure 2.6 on page 34) to produce the instantiations, because these functors are parameterized over the primitives used in the NbE module. Hence, rather than hardwiring code-generation primitives, this factorization reuses the implementation for producing both the evaluating instantiation and the residualizing instantiation. An evaluating instantiation `EFullNbE` of NbE is produced by applying the functor `makeFullNbE` to the standard evaluating structures `EExp`, `EGensym` and `ECtrl` of the signatures `EXP`, `GENSYM` and `CTRL`, respectively (Figure 2.8 on the previous page—we show the implementations of structures `EExp` and `EGensym`; for structure `ECtrl`, we use Filinski's implementation [14]). Residualizing instantiations `RFullNbE` of Full NbE and `RPureNbE` of Pure NbE result from applying the functors `makePureNbE` and `makePureNbE`, respectively, to appropriate residualizing structures `RGensym`, `RExp`, and `RCtrl` (Figure 2.9).

For example, in the structure `RExp`, the type `Exp` and the type `Var` are both instantiated with `EExp.Exp` since they are dynamic base types, and all the code-constructing functions are implemented as functions that generate 'code that constructs code'; here, to assist understanding, we have unfolded the definition of reflection (see also Example 2.3 on page 28).

```
local open EFullNbE
  infixr 5 -->
  val Ereify_aaaa_a
    = reify ((a'-->a'-->a'-->a') --> a')          (* ↓^((•→•→•→•)→•) *)
  open RPureNbE
  infixr 5 -->
  val Rreify_aaaa = reify (a'-->a'-->a'-->a')          (* ⇓^(•→•→•→•) *)
in val nf = Ereify_aaaa_a (Rreify_aaaa) end
```

The (pretty-printed) result `nf` is:

$$\lambda x_1.\ \textbf{let}\ r_2 = \mathsf{init}()\ \ r_3 = \mathsf{new}()\ \ r_4 = \mathsf{new}()\ \ r_5 = \mathsf{new}()$$
$$\textbf{in}$$
$$\underline{\lambda r_3}.\underline{\lambda r_4}.\underline{\lambda r_5}.x_1\ \underline{r_3}\ \underline{r_4}\ \underline{r_5}$$
$$\textbf{end}$$

Figure 2.10: Visualizing $\downarrow^{\bullet\to\bullet\to\bullet\to\bullet}$

With the residualizing instantiation of reification and reflection at our disposal, we now can perform visualization by following Equation (2.3) on page 40.

Example 2.18 *We show the visualization of* $\downarrow^{\bullet\to\bullet\to\bullet\to\bullet}$ *(cf. Example 2.2 on page 27) for Pure NbE. Following Equation (2.3) on page 40, we have to compute* $\downarrow^{(\bullet\to\bullet\to\bullet\to\bullet)\to\bullet}$ $(\Downarrow^{\bullet\to\bullet\to\bullet\to\bullet})$*. This is done in Figure 2.10; it is not difficult to see that the result matches the execution of the term* `reify (a' --> a' --> a' --> a')` *(see Figure 2.4 on page 32). Visualization of the reflection function is carried out similarly.*

2.4.2 An example: Church numerals

We first demonstrate the second Futamura projection with the example of the addition function for Church numerals. The definitions for the Church numeral 0_{ch}, successor s_{ch}, and the addition function $+_{\mathsf{ch}}$ in Figure 2.11 on the next page are all standard; as the types indicate, they are given as the residualizing instantiation. One can see that partially evaluating the addition function $+_{\mathsf{ch}}$ with respect to the Church numeral $n_{\mathsf{ch}} = s^n_{\mathsf{ch}}(0_{\mathsf{ch}})$ should produce a term $\lambda n_2.\lambda f.\lambda x.f^n(n_2 f x)$; by definition, this is also the functionality of a generating extension of function $+_{\mathsf{ch}}$.

The term $+_{\mathsf{ch}}$ contains no dynamic constants, hence $\overline{\overline{+_{\mathsf{ch}}}} = \overline{+_{\mathsf{ch}}} = +_{\mathsf{ch}}$. Following Equation (2.4) on page 41, we can compute an efficient generating extension $+_{\mathsf{ch}}{}^{\ddagger}$, as shown in Figure 2.11 on the next page.

```
type 'a num = ('a -> 'a) -> ('a -> 'a)                    (* Type num *)
val c0 : EExp.Exp num
  = fn f => fn x =>  x                                    (* ⌜0ch⌝ : ⌜num⌝ *)
fun cS (n: EExp.Exp num)
  = fn f => fn x => f (n f x)                             (* ⌜sch⌝ : ⌜num → num⌝ *)
fun cAdd (m: EExp.Exp num, n: EExp.Exp num)
  = fn f => fn x =>
       m f (n f x)                                        (* ⌜+ch⌝ : ⌜(num × num) → num⌝ *)

local open EFullNbE
  infixr 5 -->
  val Ereify_n_exp
  = reify  (((a' --> a') --> (a' --> a')) --> a')
                                                          (* ↓⌜num⌝→• *)
  open RPureNbE
  infixr 5 -->
  val Rreify_n_n
  = reify (((a'-->a') --> (a'-->a')) -->
           ((a'-->a') --> (a'-->a')))                     (* ⇓⌜num→num⌝ *)
in val ge_add
  = Ereify_n_exp (fn m => (Rreify_n_n (fn n =>
    cAdd (m, n))))                                        (* +ch‡ *)
end;
```

The (pretty-printed) result $+_{\mathsf{ch}}{}^{\ddagger}$ is:

$$\begin{array}{l}\lambda x_1.\ \textbf{let}\ r_2 = \mathsf{init}()\ \ r_3 = \mathsf{new}()\ \ r_4 = \mathsf{new}()\ \ r_5 = \mathsf{new}()\ \ r_7 = \mathsf{new}()\\ \quad \textbf{in}\\ \quad\quad \underline{\lambda}\, r_3.\underline{\lambda}\, r_4.\underline{\lambda}\, r_5.(x_1(\lambda\, x_6.(\underline{r_4}\ \underline{@}\ x_6)))\\ \quad\quad\quad\quad (((\underline{r_3}\ \underline{@}\ (\underline{\lambda}\, r_7.(\underline{r_4}\ \underline{@}\ \underline{r_7})))\ \underline{@}\ \underline{r_5}))\\ \quad \textbf{end}\end{array}$$

For example, applying $+_{\mathsf{ch}}{}^{\ddagger}$ to `(cS (cS (c0)))` generates

$$\lambda x_1.\lambda x_2.\lambda x_3.x_2(x_2(x_1(\lambda x_4.x_2\ x_4)x_3)).$$

Figure 2.11: Church numerals

2.4.3 The GE-instantiation

We generalize the technique of encoding a two-level term p in ML presented at the end of Section 2.2.2: We code p inside a functor

```
p_ge(structure S:STATIC structure D:DYNAMIC) = ...
```

that parameterizes over both static and dynamic constants. With suitable instantiations of the structures `S` and `D`, one can thus create not only the evaluation instantiation $\widetilde{p}$ and the residualizing instantiation $\ulcorner p \urcorner$, but also the GE-instantiation $\ulcorner\!\ulcorner p \urcorner\!\urcorner$. The instantiation table displayed in Table 2.1 summarizes how to write the components of the three kinds of instantiation functors for `S` and `D`. The table follows easily from the formal definitions of $\widetilde{\cdot}$, $\ulcorner\cdot\urcorner$ and $\ulcorner\!\ulcorner\cdot\urcorner\!\urcorner$ via $\Phi_\sim$ (Definition 2.9 on page 36), $\Phi_{\ulcorner\urcorner}$ (Definition 2.10 on page 37) and $\Phi_{\ulcorner\!\ulcorner\urcorner\!\urcorner}$ (Definition 2.17 on page 44), respectively.

		$\widetilde{\cdot}$	$\ulcorner\cdot\urcorner$	$\ulcorner\!\ulcorner\cdot\urcorner\!\urcorner$
S	$\mathsf{b}^{\mathtt{S}}$	b	b	`Exp`
	$c^{\mathtt{S}} : \tau$	c	c	$\uparrow_{\langle\ulcorner\tau\urcorner\rangle}\,\underline{\text{“}c\text{”}}$
D	$\mathsf{b}^{\mathtt{D}}$	b	`Exp`	`Exp`
	$c^{\mathtt{D}} : \tau$	c	$\uparrow_\tau\,\underline{\text{“}c\text{”}}$	$\Uparrow_\tau\,\ulcorner\langle \mathtt{VAR} \rangle\urcorner\,(\mathit{lift}_{\mathtt{string}}\text{“}c\text{”})$
	$\$_{\mathsf{b}}$	$\lambda x.x$	$\mathit{lift}_{\mathsf{b}}$	$\uparrow_{\bullet\to\bullet}\,\underline{\text{“}\mathit{lift}_{\mathsf{b}}\text{”}}$

Table 2.1: Instantiation table

Note, in particular, that $\widetilde{\cdot}$ and $\ulcorner\cdot\urcorner$ have the same instantiation for the static signature; hence we can reuse $\Phi_\sim$ for $\Phi_{\ulcorner\urcorner}$.

Example 2.19 *We revisit the function* `height`*, which appeared in Example 2.5 on page 29 and Example 2.7 on page 31. In Figure 2.12 on the following page we define the functor* `height_ge` *along with signatures* `STATIC` *and* `DYNAMIC`*. Structure* `GEStatic` *and structure* `GEDynamic` *provide the GE-instantiation for the signature* Σ^2*. The instantiation of* `height_ge` *with these structures gives* $\ulcorner\!\ulcorner \mathtt{height}_{\mathsf{ann}} \urcorner\!\urcorner$*. Applying the second Futamura projection as given in Equation (2.4) on page 41 yields*

$$
\begin{array}{l}
\lambda x_1.\ \textbf{let}\ r_2 = \mathsf{init}() \\
\qquad\quad\ \ r_3 = \mathsf{new}() \\
\qquad \textbf{in} \\
\qquad\quad \underline{\lambda\, r_3.\text{“}\mathit{mult}\text{”}}\,\underline{@}\,(\mathit{lift}_{\mathsf{real}}(\mathit{sin}\ x_1))\,\underline{@}\,\underline{r_3} \\
\qquad \textbf{end}
\end{array}
$$

2.4.4 Type specification for self-application

The technique developed so far is already sufficient to carry out visualization or the second Futamura projection, at least in an effect-free setting. Still, it requires the user to manually instantiate self-application Equation (2.3) on page 40

```
signature STATIC =                                                (* Σ^s *)
  sig
    type SReal                                                  (* real^s *)
    val sin: SReal -> SReal                                      (* sin^s *)
  end

signature DYNAMIC =                                               (* Σ^∂ *)
  sig
    type SReal                                                  (* real^s *)
    type DReal                                                  (* real^∂ *)
    val mult: DReal -> DReal -> DReal                           (* mult^∂ *)
    val lift_real: SReal -> DReal                               (* $_real *)
  end

functor height_ge(structure S: STATIC                          (* height_ann *)
                  structure D: DYNAMIC
                   sharing type D.SReal = S.SReal) =
 struct
   fun height a z = D.mult (D.lift_real (S.sin a)) z
 end

structure GEStatic: STATIC =                                    (* Φ_nn on Σ^s *)
  struct
    local open EExp EFullNbE; infixr 5 --> in
      type SReal = Exp
      val sin = reflect (a' --> a') (VAR "sin")
    end
  end

structure GEDynamic: DYNAMIC =                                  (* Φ_nn on Σ^∂ *)
  struct
    local open RExp RFullNbE; infixr 5 --> in
      type DReal = Exp
      val mult = reflect (a' --> a' --> a')
                         (VAR (EExp.STR "mult"))
      fun lift_real r = LIT_REAL r
    end
  end
                                                              (* "height_ann" *)
structure ge_height = height_ge(structure S = GEStatic
                                structure D = GEDynamic)
```

Figure 2.12: Instantiation via functors

and Equation (2.4) on page 41, as we have done for all the preceding examples. In particular, as Example 2.18 on page 47 demonstrates, one needs to use two different sets of combinators for essentially the same type ($\bullet \to \bullet \to \bullet \to \bullet$ in this case), one for the residualizing instantiation of NbE, and the other for the evaluating instantiation. It would be preferable to package the abstract formulation of Equation (2.3) on page 40 and Equation (2.4) on page 41 as program modules themselves, instead of leaving them as templates for the user.

Types are part of the input in both forms of self-application. The user of the module should specify a type τ in a way that is independent of the instantiations; it is the task of the self-application module to choose whether and where to use the residualization instantiation $(\Downarrow^\tau, \Uparrow_\tau)$ or the evaluation instantiation $(\downarrow^\tau, \uparrow_\tau)$. Since different instantiations have different types, the type argument, even in the form of an encoding of the corresponding extraction functions, cannot be abstracted over at the function level. Recall that the type-indexed functions are formed by plugging together combinators. Specifying a type, therefore, amounts to writing down how combinators should be plugged together, leaving the actual definition of the combinators (i.e., an `NBE`-structure) abstract.

To make the above idea more precise, let us consider the example of visualizing the reification functions. The specification of a type τ should consist of not only the type τ itself, but also a functor that maps a `NBE`-structure `NbE` to the appropriate instantiation of the pair $(\langle \underline{\downarrow^\tau} \rangle, \langle \underline{\uparrow_\tau} \rangle)$, which is of type τ `NbE.`rr. This suggests that the type specification should have the following dependent type:

$$\sum \tau : *. \prod \texttt{NbE} : \texttt{NBE.}(\tau\ \texttt{NbE.}rr),$$

where $\sum$ is the dependent sum formation, and $\prod$ is the dependent product formation.

We can then turn this type into a higher-order signature `VIS_INPUT` in Standard ML of New Jersey, and in turn write a higher-order functor `vis_reify` that performs visualization of the reification function (Figure 2.13 on the next page). The example visualization in Figure 2.10 on page 47 can be now carried out using the type specification given in Figure 2.14 on the next page.

2.4.5 Monomorphizing control operators

So far we have shown how to self-apply Pure TDPE. When self-applying Full TDPE, one complication arises: The implementation of Full TDPE uses control operators polymorphically in the definition of reflection, but to determine the residualizing instantiation of a constant, a fixed monomorphic type has to be determined. This section shows how to rewrite the algorithm for full TDPE such that all control operators occur monomorphically.

Let-insertion via control operators

Full TDPE treats call-by-value languages with computational effects. In this setting, *let-insertion* [3, 19] is a standard partial-evaluation technique to prevent duplicating or discarding computations that have side-effects: All computation that might have effects is bound to a variable and sequenced using the (monadic)

```
signature VIS_INPUT =                          (* Signature for a type specification *)
  sig
    type 'a vis_type                (* Type τ, parameterized at the base type *)
    functor inp(NbE: NBE) :                    (* parameterized type coding *)
      sig
        val T_enc: (NbE.Exp vis_type) NbE.rr
      end
  end

functor vis_reify (P: VIS_INPUT) =
  struct
    local
      structure eVIS                           (* Evaluating instantiation *)
              = P.inp(EFullNbE)
      structure rVIS                           (* Residualizing instantiation *)
              = P.inp(RPureNbE)
      open EFullNbE
      infixr 5 -->
    in
      val vis = reify (eVIS.T_enc --> a')                  (* ↓^τ →• (⇓^τ) *)
                      (RPureNbE.reify rVIS.T_enc)
    end
  end
```

Figure 2.13: Specifying types as functors

```
structure a2a : VIS_INPUT =                    (* A type specification *)
  struct
    type 'a vis_type = 'a->'a->'a->'a          (* τ = • → • → • → • *)
    functor inp(NbE: NBE) =                                (* NbE *)
      struct
        local open NbE  infixr 5 --> in
          val T_enc = a' --> a' --> a' --> a'              (* τ NbE.rr *)
        end
      end
    end

structure vis_a2a = vis_reify(a2a);                  (* Visualization *)
```

Figure 2.14: Type specification for visualizing $\downarrow^{\bullet\to\bullet}$

let construct. When the TDPE algorithm identifies the need to insert a let-construct, however, it usually is not at a point where a let-construct can be inserted, i.e., a code-generating expression.

Danvy [4] solves this problem using the control operators shift and reset [9], a technique that originated in continuation-based partial evaluation [24]: Intuitively speaking, the operator shift abstracts the current evaluation context up to the closest delimiter reset and passes the abstracted context to its argument, which can then invoke this delimited evaluation context just like a normal function. Formally, the semantics of shift and reset is expressed in terms of the CPS transformation (Figure 2.15; see Danvy and Filinski [9] and Filinski [14] for more details, and Danvy and Yang [13] for an operational account).

$$\begin{aligned} [\![\mathsf{shift}\ E]\!]_{\mathsf{CPS}} &= \lambda\kappa.[\![E]\!]_{\mathsf{CPS}}(\lambda f.f(\lambda v.\lambda\kappa'.\kappa'(\kappa\ v))(\lambda x.x)) \\ [\![\mathsf{reset}\langle E\rangle]\!]_{\mathsf{CPS}} &= \lambda\kappa.\kappa([\![E]\!]_{\mathsf{CPS}}(\lambda x.x)) \end{aligned}$$

Term $\langle E\rangle$, "the thunk of E", is shorthand for $\lambda().E$. The use of a thunk here delays the computation of E and avoids the need to implement reset as a macro.

Figure 2.15: The CPS semantics of shift/reset

With the help of these control operators, Danvy's treatment [4] follows the following strategy for let-insertion: (1) use reset to 'mark the boundaries' for code generation, i.e., to surround every expression that has type `Exp` and could potentially be a point where let-bindings need to be inserted;[4] (2) when let-insertion is needed, use shift to 'grab the context up to the marked boundary' and bind it to a variable k (thus k is a code-constructing context); (3) apply k to the intended return value to form the body expression of the let-construct, and then wrap it with the let-construct. The new definitions for the reification and reflection functions as given by Danvy are shown in Figure 2.16 on the next page; there are two function type constructors: a function type without effects $\tau_1 \to \tau_2$, which does not require let-insertion, and a function type with possible latent effects $\tau_1 \xrightarrow{!} \tau_2$, which does require let-insertion. We extend the type `Exp` of code representations with a constructor `LET of string * Exp * Exp` and write **let** $\underline{x} = t_1$ **in** t_2 **end** for `LET` (x, t_1, t_2); we implement a new TDPE combinator `-!>` in ML for the new type constructor $\xrightarrow{!}$.

Monomorphizing control operators In the definition of reflection $\uparrow_{\tau_1 \xrightarrow{!} \tau_2}$ for function types with latent effects, the return type (here τ_2) of the shift-expression *depends* on the type of the reflection. Hence it is not immediately amenable to be treated by TDPE itself, because during self-application, shift is regarded as a dynamic constant, whose type is needed to determine its residualizing

[4] An effect-typing system can provide a precise characterization of where reset has to be used. Roughly speaking, an operator reset encloses the escaping control effect introduced by an inner shift. See Filinski's work [15] for more details.

$$
\begin{array}{rcll}
\downarrow^{\bullet} e & = & e & \\
\downarrow^{\tau_1 \to \tau_2} f & = & \underline{\lambda\, x}.\mathsf{reset}\langle \downarrow^{\tau_2} (f(\uparrow_{\tau_1} \underline{x}))\rangle & (x \text{ is fresh}) \\
\downarrow^{\tau_1 \xrightarrow{!} \tau_2} f & = & \underline{\lambda\, x}.\mathsf{reset}\langle \downarrow^{\tau_2} (f(\uparrow_{\tau_1} \underline{x}))\rangle & (x \text{ is fresh}) \\
 & & & \\
\uparrow_{\bullet} e & = & e & \\
\uparrow_{\tau_1 \to \tau_2} e & = & \lambda\, x.\uparrow_{\tau_2} (e\, \underline{@}\, (\downarrow^{\tau_1} x)) & \\
\uparrow_{\tau_1 \xrightarrow{!} \tau_2} e & = & \lambda\, x.\mathsf{shift}\,(\lambda\, k.\underline{\mathbf{let}}\ \underline{x'} = e\, \underline{@}\ \downarrow^{\tau_1} x\ \underline{\mathbf{in}}\ \mathsf{reset}\langle k(\uparrow_{\tau_2} \underline{x'})\rangle\ \underline{\mathbf{end}}) & \\
 & & & (x' \text{ is fresh})
\end{array}
$$

Figure 2.16: TDPE with let-insertion

instantiation.

However, observe that the argument to the context k is fixed to be $\uparrow_{\tau_2} x'$; this prompts us to move this term into the context surrounding the shift-expression, and to apply k to a simple unit value $()$. Following this transformation, no information needs to be carried around, except for the transfer of the control flow.

$$
\uparrow^{\mathrm{new}}_{\tau_1 \xrightarrow{!} \tau_2} e = \lambda\, x.(\lambda\, ().\ \uparrow_{\tau_2} \underline{x'})(\mathsf{shift}\,(\lambda\, k.\underline{\mathbf{let}}\ \underline{x'} = e\, \underline{@}\ \downarrow^{\tau_1} x\ \underline{\mathbf{in}}\ \mathsf{reset}\langle k()\rangle\ \underline{\mathbf{end}}))
\qquad (x' \text{ is fresh})
$$

Now the aforementioned problem is solved, since the return type of shift is fixed to unit—the new definition is *monomorphic*.

To show that this change is semantics-preserving, we compare the CPS semantics of the original definition and the new definition.

Proposition 2.20 *The terms* $[\![\uparrow^{\mathrm{new}}_{\tau_1 \xrightarrow{!} \tau_2}]\!]_{\mathsf{CPS}}$ *and* $[\![\uparrow_{\tau_1 \xrightarrow{!} \tau_2}]\!]_{\mathsf{CPS}}$ *are* $\beta_v\eta_v$*-convertible.*

Here β_v and η_v are respectively the β and η rules in Moggi's computational lambda calculus λ_c [26], i.e., the restricted forms of the usual β rule, $(\lambda x.e')e \sim e'[x := e]$, and of the usual η rule, $\lambda x.ex \sim e$, where the expression e must be a value. These rules are sound for call-by-value languages with computational effects.

Proof: First of all, we abstract out the same computations in the two terms:

$$
\begin{array}{rcl}
B & \equiv & \lambda\, f.\underline{\mathbf{let}}\ \underline{x'} = e\, \underline{@}\ \downarrow^{\tau_1} x\ \underline{\mathbf{in}}\ f()\ \underline{\mathbf{end}} \\
R & \equiv & \uparrow_{\tau_2} \underline{x'} \\
C[\,] & \equiv & \lambda\, e.\lambda\, x.\mathbf{let}\ x' = \mathsf{new}()\ \mathbf{in}\ [\,]\ \mathbf{end}
\end{array}
$$

Then

$$\uparrow_{\tau_1 \xrightarrow{!} \tau_2} =_{\beta_v\eta_v} C[\mathsf{shift}\,(\lambda\, k.B(\lambda\,().\mathsf{reset}\langle k(R)\rangle))]$$
$$\uparrow^{\mathrm{new}}_{\tau_1 \xrightarrow{!} \tau_2} =_{\beta_v\eta_v} C[(\lambda\,().R)(\mathsf{shift}\,(\lambda\, k.B(\lambda\,().\mathsf{reset}\langle k()\rangle)))]$$

Because the CPS transformation is compositional and preserves $\beta_v\eta_v$ equivalence, it suffices to prove that the CPS transformations of the two terms enclosed by $C[\cdot]$ are $\beta_v\eta_v$-equivalent, for all terms B and R. It is a tedious but straightforward check. □

Recently, Sumii [30] pointed out that the reset in the above definition can be removed. The continuation k, being captured by shift, resets the continuation automatically when applied to an argument, which makes the reset in the above redundant, since the argument of k is a value. In contrast, the original definition still requires the reset, since the expression $\uparrow_{\tau_2} \underline{x'}$ might have latent escaping control effect, as in the case where τ_2 is of form $\tau \xrightarrow{!} \tau'$. This simplification improves the performance of TDPE and the generating extension generated by self-application.

$$\uparrow^{\mathrm{new}'}_{\tau_1 \xrightarrow{!} \tau_2} \quad e = \lambda\, x.(\lambda\,().\uparrow_{\tau_2} \underline{x'})(\mathsf{shift}\,(\lambda\, k.\underline{\mathbf{let}}\ \underline{x'} = e\,\underline{@}\downarrow^{\tau_1} x\ \underline{\mathbf{in}}\ k()\ \underline{\mathbf{end}}))$$
(x' is fresh)

Proposition 2.21 *The terms* $[\![\uparrow^{\mathrm{new}}_{\tau_1 \xrightarrow{!} \tau_2}]\!]_{\mathsf{CPS}}$ *and* $[\![\uparrow^{\mathrm{new}'}_{\tau_1 \xrightarrow{!} \tau_2}]\!]_{\mathsf{CPS}}$ *are* $\beta_v\eta_v$*-convertible.*

Proof: We proceed as in the proof of Proposition 2.20. In particular, using B, R, and $C[\,]$ introduced there, we have that

$$\uparrow^{\mathrm{new}'}_{\tau_1 \xrightarrow{!} \tau_2} =_{\beta_v\eta_v} C[(\lambda\,().R)(\mathsf{shift}\,(\lambda\, k.B(\lambda\,().k())))].$$

□

Example 2.22 *The monomorphic definitions* $\uparrow^{\mathrm{new}}_{\tau_1 \xrightarrow{!} \tau_2}$ *and* $\uparrow^{\mathrm{new}'}_{\tau_1 \xrightarrow{!} \tau_2}$ *of reflection for function types with latent effects are amenable to TDPE itself. Figure 2.17 on the following page shows the result of visualizing the reification function at the type* $(\bullet \xrightarrow{!} \bullet) \xrightarrow{!} \bullet$*. Note that both* shift *and* reset *have effects themselves; consequently TDPE has inserted let-constructs for the result of visualization. For comparison, we also show the visualization of* $(\bullet \to \bullet) \to \bullet$ *of Pure NbE, which is much more compact.*

The main difference here is the control operators used in Full TDPE, which remain in the result of self-application; later in Section 2.6, we will see how this difference affects the speedup achieved by the second Futamura projection.

Sum types Full TDPE also treats sum types using control operators; this treatment is also due to Danvy [5]. Briefly, the operator shift is used in the

Visualization of $\downarrow^{(\bullet \xrightarrow{!} \bullet) \xrightarrow{!} \bullet}$ results in:

$$
\begin{array}{l}
\lambda x_1.\mathbf{let}\ r_2 = \mathsf{init}() \\
\qquad\quad\ \ r_3 = \mathsf{new}() \\
\qquad\quad\ \ r_{11} = \mathsf{reset}\langle \mathbf{let}\ r_{10} = x_1(\lambda x_5.\mathsf{insertLet}(\underline{r_3}\ \underline{@}\ x_5))\ \mathbf{in}\ r_{10}\ \mathbf{end}\rangle \\
\quad \mathbf{in}\ \ \underline{\lambda\, r_3}.r_{11} \\
\quad \mathbf{end}
\end{array}
$$

where $\mathsf{insertLet}(E)$ abbreviates the expression

$$
\begin{array}{l}
\mathbf{let}\ x' = \mathsf{new}() \\
\quad \_\ = \mathsf{shift}(\lambda k.\mathbf{let}\ r = k()\ \mathbf{in}\ (\underline{\mathbf{let}}\ \underline{x'} = E\ \underline{\mathbf{in}}\ r\ \underline{\mathbf{end}})\ \mathbf{end}) \\
\mathbf{in}\ \ \underline{x'} \\
\mathbf{end}
\end{array}
$$

In contrast, visualization of $\downarrow^{(\bullet \to \bullet) \to \bullet}$ of Pure NbE results in:

$$
\begin{array}{l}
\lambda x_1.\mathbf{let}\ r_2 = \mathsf{init}() \\
\qquad\quad\ \ r_3 = \mathsf{new}() \\
\quad \mathbf{in}\ \ \underline{\lambda\, r_3}.x_1(\lambda x_4.\underline{r_3}\ \underline{@}\ x_4) \\
\quad \mathbf{end}
\end{array}
$$

Figure 2.17: Visualizing TDPE with let-insertion

definition of reflection function for sum types, $\uparrow_{\tau_1+\tau_2}$. As the type suggests, the return type of this function should be a value of type $\overline{\tau_1} + \overline{\tau_2}$, i.e., a value either of the form $\mathbf{inl}\ (v_1 : \overline{\tau_1})$ or $\mathbf{inr}\ (v_2 : \overline{\tau_2})$ (for some appropriate v_1 or v_2); on the other hand, both values are needed to have the complete information. Danvy's solution is to "return twice" to the context by capturing the delimited context and applying it separately to $\mathbf{inl}\ (\uparrow_{\tau_1}\ e_1)$ and $\mathbf{inr}\ (\uparrow_{\tau_2}\ e_2)$; the results are combined using a case-construct which introduces the bindings for e_1 and e_2. Danvy's definition of $\uparrow_{\tau_1+\tau_2}$ is given below:

$$
\begin{array}{l}
\uparrow_{\tau_1+\tau_2} e = \mathsf{shift}(\lambda k.\ \underline{\mathbf{case}}\ e\ \underline{\mathbf{of}}\ \underline{\mathbf{inl}}(\underline{x_1}) \Rightarrow \mathsf{reset}\langle k(\mathbf{inl}\ (\uparrow_{\tau_1}\ \underline{x_1}))\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid\ \ \underline{\mathbf{inr}}(\underline{x_2}) \Rightarrow \mathsf{reset}\langle k(\mathbf{inr}\ (\uparrow_{\tau_2}\ \underline{x_2}))\rangle\ ) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (x_1, x_2 \text{ are fresh})
\end{array}
$$

where `Exp` has been extended with constructors for a case distinction and injection functions in the obvious way. Again, the return type of the shift-expression in the above definition is not fixed; an alternative definition is needed to allow self-application.

Following the same analysis as before, we observe that the arguments to k must be one of the two possibilities, $\mathbf{inl}\ (\uparrow_{\tau_1}\ e_1)$ and $\mathbf{inr}\ (\uparrow_{\tau_2}\ e_2)$, so the information to be passed through the continuation is just the binary choice between the left branch and the right branch. We can thus move these two fixed arguments into the context and replace them with the booleans **true** and **false**

as the argument to continuation k (again, Sumii's remark on the redundancy of reset in the program after change applies, and we have dropped the unnecessary occurrences of reset):

$$\begin{aligned}\uparrow^{\mathrm{new}}_{\tau_1+\tau_2} e = \quad & \textbf{if } \mathsf{shift}(\lambda k.\ \underline{\textbf{case}}\ e\ \underline{\textbf{of}}\ \underline{\textbf{inl}}(\underline{x_1}) \Rightarrow k\ \textbf{true} \\ & \qquad\qquad\qquad\qquad \mid\ \underline{\textbf{inr}}(\underline{x_2}) \Rightarrow k\ \textbf{false}\ ) \\ & \textbf{then inl}\ (\uparrow_{\tau_1} \underline{x_1})\ \textbf{else inr}\ (\uparrow_{\tau_2} \underline{x_2}) \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (x_1, x_2 \text{ are fresh})\end{aligned}$$

The use of shift is instantiated with the fixed boolean type. Again, we check that this change does not modify the semantics.

Proposition 2.23 $[\![\uparrow^{\mathrm{new}}_{\tau_1+\tau_2}]\!]_{\mathsf{CPS}}$ *and* $[\![\uparrow_{\tau_1+\tau_2}]\!]_{\mathsf{CPS}}$ *are* $\beta_v\eta_v$*-convertible.*

Using $\uparrow^{\mathrm{new}}_{\tau_1+\tau_2}$ and $\uparrow^{\mathrm{new}'}_{\tau_1 \xrightarrow{!} \tau_2}$ instead of the original definitions provides us with an algorithm for Full TDPE that is amenable to self-application. In the following section, we use self-application of Full TDPE for compiler generation.

2.5 Generating a compiler for Tiny

It is well known that partial evaluation allows compilation by specializing an interpreter with respect to a source program. TDPE has been used for this purpose in several instances [4, 5, 11, 12]. Having implemented the second Futamura projection, we can instead *generate* a compiler as the generating extension of an interpreter.

One of the languages for which compilation with TDPE has been studied is *Tiny* [4, 27], a prototypical imperative language. As outlined in Section 2.2.2, a functor `tiny_pe(D:DYNAMIC)` is used to carry out type-directed partial evaluation in a convenient way. This functor provides an interpreter `meaning` that is parameterized over all dynamic constructs. Appendix 2.B.1 gives an overview of Tiny and type-directed partial evaluation of a Tiny interpreter. Compiling Tiny programs by partially evaluating the interpreter `meaning` corresponds to running the trivial generating extension `meaning`†.

Following the development in Section 2.4.3, we proceed in three steps to generate a Tiny compiler:

1. Rewrite `tiny_pe` into a functor `tiny_ge(S: STATIC D: DYNAMIC)` in which `meaning` is also parameterized over all static constants and base types.

2. Give instantiations of `S` and `D` as indicated by the instantiation table in Table 2.1 on page 49, thereby creating the GE-instantiation "`meaning`".

3. Perform the second Futamura projection; this yields the efficient generating extension `meaning`‡, i.e., a Tiny compiler.

Appendix 2.B.2 describes these steps in more detail.

Tiny was the first substantial example we treated; nevertheless we were done within a day—none of the three steps described above is conceptually difficult. They can be seen as a methodology for performing the second Futamura projection in TDPE on a binding-time-separated program.

Although conceptually simple, the first of the three aforementioned steps is somewhat tedious:

- Every construct that is not handled automatically by TDPE has to be parameterized over. This is not a problem for user-defined constants, but is a problem for ML-constructs like recursion and case-distinctions over recursive data types. Both have to be rewritten, using fixed-point operators and elimination functions, respectively.
- For every occurrence of a constant in the program, its monotype has to be determined; constants used at more than one monotype give rise to several instances. This is a consequence of performing *type-directed* partial evaluation; for the second Futamura projection, every constant is instantiated with a code-generation function, the form of which depends on the exact type of the constant in question.

Because the Tiny interpreter we started with was already binding-time separated, we did not have to perform the binding-time analysis needed when starting from scratch. Our experience with TDPE, however, shows that performing such a binding-time analysis is relatively easy, because

- TDPE restricts the number of constructs that have to be considered, since functions, products and sums do not require binding-time annotations, and
- TDPE uses the ML type system: Type checking checks the consistency of the binding-time annotations.

2.6 Benchmarks

2.6.1 Experiments and results

In Section 2.3 we claimed that the specialized generating extension $p^{\ddagger}$ of a program p produced by the second Futamura projection for TDPE is, in general, more efficient than the trivial generating extension $p^{\dagger}$. In order to assess how much more efficient $p^{\ddagger}$ is than $p^{\dagger}$, we performed benchmarks for $+_{\mathsf{ch}}$ (Section 2.4.2) and the Tiny interpreter (Section 2.5).

The benchmarks were performed on a 250 MHz Silicon Graphics O_2 workstation using Standard ML of New Jersey version 110.0.3. We display the results in Table 2.2 on the facing page. In each row of the table, we compare the time it takes to specialize the subject program p with respect to the static input s using two different generating extensions: (1) the trivial generating extension $p^{\dagger}$ (i.e., directly running TDPE on program p), and (2) the specialized generating extension $p^{\ddagger}$ (i.e., running the result of the second Futamura projection). We calculate the speedup as the ratio of their running times.

program p	static inp. s	specialization time with $p^\dagger$ (s)	specialization time with $p^\ddagger$ (s)	Speedup (ratio)
`meaning`	`factorial`	261.2	194.9	1.34
$\texttt{meaning}_{\mathsf{orig}}$	`factorial`	169.5	99.2	1.71
$+_{\mathsf{ch}}$	80_{ch}	58.45	19.95	2.93

Table 2.2: Benchmarks: time of specializations (1,000,000 repeated executions)

The first row compares the compilers derived from the interpreter `meaning` (see Section 2.5 and Appendix 2.B); the result shows a speedup of 1.34 for compiling the factorial function. One might wonder, however, whether there is any real gain in using the second Futamura projection: The changes that are necessary to provide the GE-instantiation of `meaning` (replace built-in pattern-matching and recursive function definition of ML with user-defined fixed-point operators and case operators, respectively—see Section 2.5) slow down both direct compilation with TDPE and compilation using the specialized generating extension. In fact, as the table's second row shows, direct compilation with the 'original' interpreter $\texttt{meaning}_{\mathsf{orig}}$, i.e., an instantiation of `tiny_pe` rather than `tiny_ge` (cf. Sections 2.2.2 and 2.4.3), runs even faster than the specialized generating extension $\texttt{meaning}^\ddagger$.

We can do better by replacing the user-defined fixed point operators and case operators in the result program $\texttt{meaning}^\ddagger$ with the built-in constructs.[5] This yields a program that can be understood as the specialized generating extension of the program $\texttt{meaning}_{\mathsf{orig}}$, and we thus call it $\texttt{meaning}_{\mathsf{orig}}{}^\ddagger$. The second row of Table 2.2 shows that running $\texttt{meaning}_{\mathsf{orig}}{}^\ddagger$ gives a speedup of 1.71 over running the original program $\texttt{meaning}_{\mathsf{orig}}$. The speedup over the direct compilation using the original interpreter here is, in practice, more relevant than the speedup of the benchmark shown in the first row.

The benchmark in the third row compares the generating extensions of an effect-free function, the addition function $+_{\mathsf{ch}}$ for Church numerals. Because the function is free of computational effects (we assume that its argument function is also effect-free), we can specialize Pure TDPE instead of Full TDPE in the second Futamura projection. The speedup of running the specialized generating extension over direct partial evaluation is consistently around 3 (shown with Church numeral 80_{ch}).

2.6.2 Analysis of the result

Overall, the speedup of the second Futamura projection with TDPE is disappointing compared to the typical order-of-magnitude speedup achievable in

[5] Removing the user-defined fixed point operator and case operators can be carried out automatically by (1) incorporating TDPE with patterns as generated bindings, and (2) systematically changing the residualizing instantiations for the fixed point and case operators used. Danvy and Rhiger [11] achieved a similar effect in TDPE for Scheme, using Scheme macros.

traditional partial evaluation [22]. This, on the other hand, reflects the high efficiency of TDPE, which carries out static computations by evaluation rather than symbolic manipulation. Turning symbolic manipulation (i.e., interpretation) into evaluation is one of the main goals one hopes to achieve by specializing a syntax-directed partial evaluator. Since TDPE does not have much interpretive overhead in the first place, the speedup is bound to be lower.

Logically, the next question to ask—for a better understanding of how and when the second Futamura projection could effectively speedup the TDPE process—is what cost of TDPE can or cannot be removed by using the self-application. The higher-order nature of the TDPE algorithm blurs the boundaries between the various components that contribute to the running time of the specialization; we can only roughly divide the cost involved in performing TDPE as follows:

1. Cost due to computation in the extraction function $\downarrow^\tau$, namely function invocations of reification and reflection for subtypes of τ, name and code generation and, in the case of Full TDPE, the use of control operators shift and reset.

2. Cost due to computation in the residualizing instantiation $\ulcorner ps \urcorner$ of input program and static input, namely, apart from static computation, the invocation of reflection by code-generation versions of dynamic constants (see Section 2.2.1).

3. Cost due to reducing extra redexes formed by the interaction of $\downarrow^\tau$ and $\ulcorner ps \urcorner$ in $\downarrow^\tau \ulcorner ps \urcorner$.

Of the costs due to computation in the extraction function, only the one caused by function invocations can be eliminated, which amounts to function inlining. All other computations have to be performed at specialization time. Similarly, for the cost associated with the residualizing instantiation, inlining can be performed for the code-generation versions of dynamic constants and their calls to the reflection function. Finally, the extra redexes formed by the interaction of the extraction function and the residualizing instantiation can be partly reduced by the specialization.

In Full TDPE, the somewhat time-consuming control operators dominate the cost of extraction algorithm; the low speedup of specializing Full TDPE (the first two benchmarks) as opposed to that of specializing Pure TDPE (the third benchmark), we think, are mainly due to the fact that these control operators cannot be eliminated. Furthermore, in the case of the Church addition function, the program is a higher-order pure λ-term, which usually "mixes well" with the extraction function, in the sense that many extra redexes are formed by their interaction.

Do certain implementation-related factors, such as the global optimizations of the ML compiler we used and the fact that we are working in a typed setting, give positive contribution to the speedup? In our opinion, the help is minimal, if not negative. First, the specialization carried out by the self-application with

respect to a trivial BTA (Section 2.3.1) has an effect similar to a good global inliner. Therefore, the global optimization of the ML compiler, especially the inlining optimization, should only reduce the potential speedup of the specialization. Second, working in a typed setting does complicates the type specification and the parameterization (Section 2.4.4), but it does not incur extra cost at runtime when using TDPE. Indeed, the instantiation through ML functors happens at compile time. Furthermore, the need to parameterize over built-in constructs such as fixed point operators and pattern matching is present also in an untyped setting.

2.7 Conclusions and issues

We have adapted the underlying concept of the second Futamura projection to TDPE and derived an ML implementation for it. By treating several examples, among them the generation of a compiler from an interpreter, we have examined the practical issues involved in using our implementation for deriving generating extensions of programs.

To hand-write a cogen and to formally prove its correctness at the same time, one possibility is to start with a partial evaluator and rewrite it into the desired generating extension in several steps, such as the use of higher-order abstract syntax and deforestation in Thiemann's work [31]. Correctness follows from showing the correctness of the partial evaluator and the correctness of each of these steps. In contrast, for generating extensions produced with the second Futamura projection, the implementation is produced automatically, and correctness follows immediately from the correctness of the partial evaluator. Often, however, this conceptual simplicity is compromised by (1) the complications in using self-application, and (2) the need to make the partial evaluator self-applicable and prove the necessary changes to be meaning preserving. In the case of TDPE, the implementation effort for writing the GE-instantiation of the object program is similar in level to that of the hand-written cogen approach, but the only change to the TDPE algorithm itself is the transformation described and proven correct in Section 2.4.5.

The third Futamura projection states that specializing a partial evaluator with respect to itself yields an efficient generating-extension generator. The type-indexed nature of TDPE makes it challenging to implement the third Futamura projection directly in ML. If it can be done, our experience with the second Futamura projection suggests that only an insignificant speedup would be obtained.

At the current stage, our contribution seems to be more significant at a conceptual level, since the speedup achieved by using the generated generating extensions is rather modest. However we observed that a higher speedup can be achieved for more complicated type structures, especially in a setting with no or few uses of computational effects; this suggests that our approach to the second Futamura projection using TDPE might find more practical applications in, e.g., the field of type theory and theorem proving.

The technical inconveniences mentioned in Section 2.5 are clearly an obstacle for using the second Futamura projection for TDPE (and, to a lesser extent, for using TDPE itself). A possible solution is to implement a translator from the two-level language into ML, thus handling the mentioned technicalities automatically. Of course, such an approach would sacrifice the flexibility of TDPE of allowing the use of all language constructs in the static part of the subject program. Even so, TDPE would still retain a distinct flavor when compared to traditional partial evaluation techniques: Only those constructs not handled automatically by TDPE, i.e., constants, need to be binding-time annotated; other constructs, such as function application and function abstraction, always follow their standard typing rules from typed lambda calculi. This simplifies the binding-time analysis considerably and often makes binding-time improvements, e.g, eta-expansion, unnecessary, which was one of the original motivations of TDPE [5, 10].

Acknowledgments

At an early stage both Olivier Danvy and Morten Rhiger [29] independently implemented a similar version of the second Futamura projection, thus providing further stimulation for our work. Andrzej Filinski's formal treatment of TDPE proved to be invaluable for understanding the second Futamura projection for TDPE. Eijiro Sumii pointed out how the monomorphizing transformations can be improved (see Section 2.4.5).

We are grateful to Daniel Damian, Olivier Danvy, Andrzej Filinski, Lasse R. Nielsen, Morten Rhiger, our anonymous referees from HOSC and PEPM'00, and our editor Julia Lawall for their numerous constructive comments.

2.A Notation and symbols

Font Conventions

p, s, d, ...	terms (one-level or two-level)	22
x, y, z, ...	variable names	22
$\underline{x}$, $\underline{@}$, $\underline{\textbf{let}}$,...	constructors for code representation (of type `Exp`)	26

Language

Σ	signature	34
$\mathcal{I}$	interpretation	34
$(\Sigma, \mathcal{I})$	language specification	34
$t : \sigma$ or $\Gamma \vdash_\Sigma t : \sigma$	typing judgment	34
$[\![\cdot]\!]$, $[\![\cdot]\!]^{\mathcal{I}}$	meaning function	22,35
$t[x := t']$	substitution of t' for x in t	
Φ	instantiation	35

$t\{\Phi\}$	application of Φ to t	35

Two-level language

$\Sigma^2 = \Sigma^{\mathsf{s}},\Sigma^{\mathsf{d}}$	two-level signature with static part Σ^{s} and dynamic part Σ^{d}	35
c^{s}, b^{s}, ...	static constants and base-types (part of Σ^{s})	35
c^{d}, b^{d}, ...	dynamic constants and base-types (part of Σ^{d})	35
$\$_{\mathsf{b}}$	lifting function on base-type b (part of Σ^{d})	35
$\mathcal{I}^{\mathsf{s}}$	interpretation of static signature Σ^{s}	35
$\mathcal{I}^{\mathsf{d}}$	interpretation of dynamic signature Σ^{d}	35
$PL^2 = (\Sigma^2, \mathcal{I}^{\mathsf{s}})$	two-level language (fixing only the interpretation of Σ^{s})	35
$PL = (\Sigma^{PL}, \mathcal{I}^{PL})$	one-level language associated with PL^2	35
$NF(e)$	static normal-form of two-level term e	36
t_{ann}	binding-time annotated term (a two-level term)	38
$\langle\underline{\ .\ }\rangle$	trivial binding-time annotation (defined by $\Phi_{\langle\rangle}$)	40,41
$\tilde{\cdot}$	evaluating instantiation (defined by $\Phi_{\sim}$)	36
$\overline{\cdot}$	residualizing instantiation (defined by $\Phi_{\neg}$)	37
$\overline{\cdot}^{\cdot}$	GE-instantiation (defined by Φ_{m})	44

PE-specific notation

PE	code of a partial evaluator	22
p_s	result of specializing program p to input s	22
$p^\dagger$	trivial generating extension of program p	40
$p^\ddagger$	efficient generating extension of program p	41

TDPE-specific notation

$\downarrow^\tau$	reification function at type τ	27
$\uparrow_\tau$	reflection function at type τ	27
$\bullet$	abbreviation for any base type	27
$\Downarrow^\tau$	abbreviation for $\overline{\langle\downarrow^\tau\rangle}$	40
$\Uparrow_\tau$	abbreviation for $\overline{\langle\uparrow_\tau\rangle}$	40

ML function symbols

init	initializes the name generator	45
new	generates a new name	45
shift, reset	control operators	53

2.B Compiler generation for Tiny

2.B.1 A binding-time-separated interpreter for Tiny

Paulson's Tiny language [27] is a prototypical imperative language—the BNF of its syntax is given in Figure 2.18 on the following page. Figure 2.19 displays

the factorial function coded in Tiny.

```
program     ::=  block declaration in command end

declaration ::=  identifier*

command     ::=  skip
                 | command ; command
                 | identifier := expression
                 | if expression then command else command
                 | while expression do command end

expression  ::=  literal
                 | identifier
                 | (expression primop expression)

identifier  ::=  a string

literal     ::=  an integer

primop      ::=  + | - | * | < | =
```

Figure 2.18: BNF of Tiny programs

```
block res val aux in
  aux := 1;
  while (0 < val) do
    aux := (aux * val);
    val := (val - 1)
  end;
  res := aux
end
```

Figure 2.19: Factorial function in Tiny

Experiments in type-directed partial evaluation of a Tiny interpreter with respect to a Tiny program [4, 5] used an ML implementation of a Tiny interpreter (Figure 2.20 on page 68): For every syntactic category a meaning function is defined—see Figure 2.21 on page 69 for the ML data type representing Tiny syntax. The meaning of a Tiny program is a function from store to store; the interpreter takes a Tiny program together with a initial store and, provided it terminates on the given program, returns a final store. Compilation by partially evaluating the interpreter with respect to a program thus results in the ML code of the store-to-store function denoted by the program.

Performing a binding-time analysis on the interpreter (under the assumptions that the input program is static and the input store is dynamic) classifies all the constants in the bodies of the meaning functions as dynamic; literals have to be lifted. As described at the end of Section 2.2.3, the interpreter is implemented within a functor that abstracts over all dynamic constants (for example `cond`, `fix` and `update` in `mc`). This allows one to easily switch between the evaluating instantiation `meaning` and the residualizing instantiation `'meaning'`. For the evaluating instantiation we simply instantiate the functor with the actual constructs, for example

```
fun cond (b, kt, kf, s) = if b <> 0 then kt s else kf s

fun fix f x             = f (fix f) x
```

For the residualizing instantiation `'meaning'` we instantiate the dynamic constants with code-generation functions; as pointed out in Example 2.3 on page 28 and made precise in Definition 2.10 on page 37, reflection can be used to write code-generation functions:

```
fun cond e  = reflect (rrT4 (a', a' -!> a', a' -!> a', a')
                       -!> a')
                      (VAR "cond") e
fun fix f x = reflect (((a' -!> a') --> (a' -!> a')) -->
                       (a' -!> a'))
                      (VAR "fix") f x
```

2.B.2 Generating a compiler for Tiny

As mentioned in Section 2.5, we derive a compiler for Tiny in three steps:

1. rewrite `tiny_pe` into a functor `tiny_ge(S:STATIC D:DYNAMIC)` such that the interpreter `meaning` is also parameterized over all static constants and base types
2. give instantiations of `S` and `D` as indicated by the instantiation table in Table 2.1 on page 49, thereby creating the GE-instantiation `'meaning'`
3. use the GE-instantiation `'meaning'` to perform the second Futamura projection

The following two sections describe the first two steps in more detail. Once we have a GE-instantiation, the third step is easily carried out with the help of an interface similar to the one for visualization described in Section 2.4.4.

2.B.3 "Full parameterization"

Following Section 2.4.3 we re-implement the interpreter inside a functor to parameterize over *both* static and dynamic base types and constants. Note, however, that the original implementation of Figure 2.20 on page 68 makes use of

recursive definitions and case distinctions; both constructs cannot be parameterized over directly. Hence we have to express recursive definitions with a fixed point operator and case distinctions with appropriate elimination functions. Consider for example case distinction over `Expression`; Figure 2.22 on page 69 shows the type of the corresponding elimination function.

The resulting implementation is sketched in Figure 2.23 on page 70. The recursive definition is handled by a top-level fixed point operator, and all the case distinctions have been replaced with a call to the corresponding elimination function.

Now that we are able to parameterize over every construct, we enclose the implementation in a functor as shown in Figure 2.24 on page 71. The functor takes two structures; their respective signatures `STATIC` and `DYNAMIC` declare names for all base types and constants that are used statically and dynamically, respectively. A base type (for example `int`) may occur both statically ($\texttt{int}^{\texttt{s}}$) and dynamically ($\texttt{int}^{\texttt{d}}$)—in this case two distinct names (for example `Int_s` and `Int_d`) have to be used.

As mentioned in Section 2.5, the monotype of every instance of a constant appearing in the interpreter has to be determined. It is these monotypes that must be declared in the signatures `STATIC` and `DYNAMIC`. Figure 2.25 on page 71 shows a portion of signature `STATIC`: The polymorphic type of `caseExpression` (Figure 2.22 on page 69) gives rise to a type abbreviation `case_Exp_type`, which can be used to specify the types of the different instances of `caseExpression`. Note that if a static polymorphic constant is instantiated with a type that contains dynamic base types—like `Int_d` in the case of `caseExpression`—then these dynamic base types have to be included in the signature `STATIC` of static constructs.[6] For base types that occur both in signatures `STATIC` and `DYNAMIC`, sharing constraints have to be declared in the interface of functor `tiny_ge` (Figure 2.24 on page 71).

Finding the monotypes for the various instantiations of constants in the interpreter can be facilitated by using the type-inference mechanism of ML: We transcribe the output of ML type inference into a type specification by hand. This transcription is straightforward, because the type specifications of TDPE and the output of ML type inference are very much alike.

2.B.4 The GE-instantiation

After parameterizing the interpreter as described above, we are in a position to either run the interpreter by using its evaluating instantiation (see Definition 2.9 on page 36), perform type-directed partial evaluation by employing the residualizing instantiation (Definition 2.10 on page 37), or carry out the second Futamura projection with the GE-instantiation (Definition 2.17 on page 44).

[6] Note that static base types appear also in the signature of dynamic constructs, because we make the lifting functions part of the latter. However there is a conceptual difference: in a two-level language, it is natural that the dynamic signature has dependencies on the static signature, whereas the static signature should not depend on the dynamic signature.

Section 2.4.3 shows how the static and dynamic constructs have to be instantiated in each case. For the GE-instantiation, all base types become `Exp`; static and dynamic constants are instantiated with code-generation functions. The latter are constructed using the evaluating and the residualizing instantiation of reflection, respectively. Because the signatures `STATIC` and `DYNAMIC` hold the precise type at which each constant is used, it is purely mechanical to write down the structures needed for the GE-instantiation.

```
fun meaning p store =
  let fun mp (PROGRAM (vs, c)) s                              (* program *)
        = md vs 0 (fn env => mc c env s)
      and md [] offset k                                      (* declaration *)
        = k (fn i => ~1)
        | md (v :: vs) offset k
        = (md vs (offset + 1)
           (fn env => k (fn i => if v = i
                                 then offset
                                else env i)))
      and mc (SKIP) env s                                     (* command *)
        = s
        | mc (SEQUENCE(c1, c2)) env s
        = mc c2 env (mc c1 env s)
        | mc (ASSIGN(i, e)) env s
        = update (lift_int (env i), me e env s, s)
        | mc (CONDITIONAL(e, c_then, c_else)) env s
        = cond (me e env s,
                mc c_then env,
                mc c_else env,
                s)
        | mc (WHILE(e, c)) env s
        = fix (fn w => fn s
               => cond (me e env s,
                        fn s => w (mc c env s),
                        fn s => s,
                        s) ) s
      and me (LITERAL l) env s                                (* expression *)
        = lift_int l
        | me (IDENTIFIER i) env s
        = fetch (lift_int (env i), s)
        | me (PRIMOP2(rator, e1, e2)) env s
        = mo2 rator (me e1 env s) (me e2 env s)
      and mo2 b v1 v2                                         (* primop *)
        =
        case b of
            Bop_PLUS => add (v1, v2)
          | Bop_MINUS => sub (v1, v2)
          | Bop_TIMES => mul (v1, v2)
          | Bop_LESS => lt (v1, v2)
          | Bop_EQUAL => eqi (v1, v2)
  in
    mp p store
  end
```

Figure 2.20: An interpreter for Tiny

```
type Identifier = string

datatype
  Program =                                   (* program and declaration  *)
    PROGRAM of Identifier list * Command
and
  Command =                                                (* command *)
    SKIP                                                     (* skip *)
  | SEQUENCE of Command * Command                              (* ; *)
  | ASSIGN of Identifier * Expression                          (* := *)
  | CONDITIONAL of Expression * Command * Command              (* if *)
  | WHILE of Expression * Command                            (* while *)
and
  Expression =                                            (* expression *)
    LITERAL of int                                          (* literal *)
  | IDENTIFIER of Identifier                              (* identifier *)
  | PRIMOP2 of Bop * Expression * Expression                (* primop *)
and
  Bop =                                                     (* primop *)
    Bop_PLUS                                                    (* + *)
  | Bop_MINUS                                                   (* - *)
  | Bop_TIMES                                                   (* * *)
  | Bop_LESS                                                    (* < *)
  | Bop_EQUAL                                                   (* = *)
```

Figure 2.21: Datatype for representing Tiny programs

```
val case_Expression
  : Expression -> ((Int_s -> 'a) *
                   (Identifier -> 'a) *
                   (Bop * Expression * Expression -> 'a)
                   ) -> 'a
```

Figure 2.22: An elimination function for expressions

```
fun meaning p store =
  let val (mp, _, _, _, _) =
    fix5
    (fn (mp, md, mc, me, mo2) =>
     let fun mp' prog                                          (* program *)
           = ...
         and md' idList                                      (* declaration *)
           = ...
         and mc' c                                            (* command *)
           = (case_Command c
              (                                        (* mc (SKIP) env s *)
               fn _ => fn env => fn s
               => s,
                                              (* mc (SEQUENCE(c1, c2)) env s *)
               fn (c1, c2) => fn env => fn s
               => mc c2 env (mc c1 env s),
                                                   (* mc (ASSIGN(i, e)) env s *)
               fn (i, e) => fn env => fn s
               => update (lift_int (env i), me e env s, s),
                          (* mc (CONDITIONAL(e,c_then,c_else)) env s *)
               fn (e, c_then, c_else) => fn env => fn s
               => cond (me e env s,
                        mc c_then env,
                        mc c_else env,
                        s),
                                                   (* mc (WHILE (e, c)) env s *)
               fn (e, c) => fn env => fn s
               => fix (fn w
                       => fn s
                       => cond (me e env s,
                                fn s => w (mc c env s),
                                fn s => s,
                                s)) s
              ))
         and me' e                                          (* expression *)
           = (case_Expression e (...))
         and mo2' bop                                         (* primop *)
           = (case_Bop bop (...))
     in
       (mp', md', mc', me', mo2')
     end)
  in
    mp p store
  end
```

Figure 2.23: A fully parameterizable implementation

```
functor tiny_ge (structure S : STATIC
                 structure D : DYNAMIC
                   sharing type S.Int_s = D.Int_s
                           ⋮ )=
  struct
    local open S D
    in
      fun meaning p store
        = ...
    end
  end
```

Figure 2.24: Parameterizing over both static and dynamic constructs

```
⋮
type 'a case_Exp_type                                (* Type abbreviation *)
  = Expression -> ((Int_s -> 'a) *
                   (Identifier -> 'a) *
                   (Bop * Expression * Expression -> 'a)
                   ) -> 'a

type case_Exp_res_type                               (* Result type *)
  = (Identifier -> Int_s) -> sto -> Int_d

⋮
                        (* Declaration of elimination function for expressions *)
val case_Expression: case_Exp_res_type case_Exp_type

⋮
```

Figure 2.25: Excerpts from signature `STATIC`

Bibliography

[1] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In Albert R. Meyer, editor, *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science*, pages 203–213, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[2] Lars Birkedal and Morten Welinder. Hand-writing program generator generators. In Manuel Hermenegildo and Jaan Penjam, editors, *Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 198–214, Madrid, Spain, September 1994. Springer-Verlag.

[3] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[4] Olivier Danvy. Pragmatic aspects of type-directed partial evaluation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation, Proceedings*, number 1110 in Lecture Notes in Computer Science, pages 73–94. Springer-Verlag, 1996.

[5] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

[6] Olivier Danvy. A simple solution to type specialization. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of ICALP '98*, number 1443 in Lecture Notes in Computer Science, pages 908–917, Aalborg, Denmark, 1998. Springer-Verlag.

[7] Olivier Danvy. Type-directed partial evaluation. In *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag. xtended version available as BRICS technical report LN-98-3.

[8] Olivier Danvy and Peter Dybjer, editors. *Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98,* (Gothenburg, Sweden, May 8–9, 1998), number NS-98-8 in Note Series, Department of Computer Science, University of Aarhus, May 1998. BRICS.

[9] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[10] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 8(6):730–751, 1996.

[11] Olivier Danvy and Morten Rhiger. Compiling actions by partial evaluation, revisited. Technical Report BRICS-RS-98-13, BRICS, Department of Computer Science, University of Aarhus, June 1998.

[12] Olivier Danvy and René Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. In Herbert Kuchen and Doaitse Swierstra, editors, *Eighth International Symposium on Programming Language Implementation and Logic Programming*, number 1140 in Lecture Notes in Computer Science, pages 182–197, Aachen, Germany, September 1996. Springer-Verlag. Extended version available as BRICS technical report RS-96-13.

[13] Olivier Danvy and Zhe Yang. An operational investigation of the CPS hierarchy. In S. Doaitse Swierstra, editor, *Proceedings of the Eighth European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, pages 224–242, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[14] Andrzej Filinski. Representing monads. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, January 1994. ACM Press.

[15] Andrzej Filinski. Representing layered monads. In Alex Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, January 1999. ACM Press.

[16] Andrzej Filinski. A semantic account of type-directed partial evaluation. In Gopalan Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 378–395, Paris, France, September 1999. Springer-Verlag.

[17] Yoshihito Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*,

12(4):363–397, 1999. Reprinted from Systems · Computers · Controls 2(5), 1971.

[18] Bernd Grobauer and Zhe Yang. Source code for the second Futamura projection for type-directed partial evaluation in ML, 2000. Available from `http://www.brics.dk/~tdpe/second_FP/sources.tgz`.

[19] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7:507–541, 1997. Extended version available as BRICS technical report RS-96-34.

[20] Carsten K. Holst and John Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, 4^{th} Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218. Glasgow University, 1991.

[21] Neil D. Jones. Challenging problems in partial evaluation and mixed computation. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 1–14. North-Holland, 1988.

[22] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International Series in Computer Science. Prentice-Hall International, 1993.

[23] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[24] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In Carolyn L. Talcott, editor, *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, pages 227–238, Orlando, Florida, June 1994. ACM Press.

[25] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[26] Eugenio Moggi. Computational lambda-calculus and monads. In Rohit Parikh, editor, *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[27] Lawrence C. Paulson. Compiler generation from denotational semantics. In Bernard Lorho, editor, *Methods and Tools for Compiler Construction*, pages 219–250. Cambridge University Press, 1984.

[28] Morten Rhiger. Deriving a statically typed type-directed partial evaluator. In Olivier Danvy, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99), Proceedings*, BRICS technical report BRICS-NS-99-1, pages 25–29, Department of Computer Science, University of Aarhus, 1999. BRICS.

[29] Morten Rhiger. Run-time code generation for type-directed partial evaluation. Progress report, BRICS PhD School, University of Aarhus. Available at `http://www.brics.dk/~mrhiger`, 1999.

[30] Eijiro Sumii, 2000. Email exchange, February 2000.

[31] Peter Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, 1999.

[32] Zhe Yang. Encoding types in ML-like languages. In Paul Hudak and Christian Queinnec, editors, *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 289–300, Baltimore, Maryland, September 1998. ACM Press. Extended version available as BRICS technical report RS-98-9.

Chapter 3

A Unifying Approach to Goal-Directed Evaluation

Abstract

Goal-directed evaluation, as embodied in Icon and Snobol, is built on the notions of backtracking and of generating successive results, and therefore it has always been something of a challenge to specify and implement. In this article, we address this challenge using computational monads and partial evaluation.

We consider a subset of Icon and we specify it with a monadic semantics and a list monad. We then consider a spectrum of monads that also fit the bill, and we relate them to each other. For example, we derive a continuation monad as a Church encoding of the list monad. The resulting semantics coincides with Gudeman's continuation semantics of Icon.

We then compile Icon programs by specializing their interpreter (i.e., by using the first Futamura projection), using type-directed partial evaluation. Through various back ends, including a run-time code generator, we generate ML code, C code, and OCaml byte code. Binding-time analysis and partial evaluation of the continuation-based interpreter automatically give rise to C programs that coincide with the result of Proebsting's optimized compiler.

3.1 Introduction

Goal-directed languages combine expressions that can yield multiple results through backtracking. Results are generated one at a time: an expression can either succeed and generate a result, or fail. If an expression fails, control is passed to a previous expression to generate the next result, if any. If so, control

is passed back to the original expression in order to try whether it can succeed this time. Goal-directed programming specifies the order in which subexpressions are retried, thus providing the programmer with a succint and powerful control-flow mechanism. A well-known goal-directed language is Icon [11].

Backtracking as a language feature complicates both semantics and implementation. Gudeman [13] gives a continuation semantics of a goal-directed language; continuations have also been used in implementations of languages with control structures similar to those of goal-directed evaluation, such as Prolog [3, 15, 30]. Proebsting and Townsend, the implementors of an Icon compiler in Java, observe that continuations can be compiled into efficient code [1, 14], but nevertheless dismiss them because "[they] are notoriously difficult to understand, and few target languages directly support them" [23, p.38]. Instead, their compiler is based on a translation scheme proposed by Proebsting [22], which is based on the four-port model used for describing control flow in Prolog [2]. Icon expressions are translated to a flow-chart language with conditional, direct and indirect jumps using templates; a subsequent optimization which, amongst other things, reorders code and performs branch chaining, is necessary to produce compact code. The reference implemention of Icon [12] compiles Icon into byte code; this byte code is then executed by an interpreter that controls the control flow by keeping a stack of expression frames.

In this article, we present a unified approach to goal-directed evaluation:

1. We consider a spectrum of semantics for a small goal-directed language. We relate them to each other by deriving semantics such as Gudeman's [13] as instantiations of one generic semantics based on computational monads [21]. This unified approach enables us to show the equivalence of different semantics simply and systematically. Furthermore, we are able to show strong conceptual links between different semantics: Continuation semantics can be derived from semantics based on lists or on streams of results by Church-encoding the lists or the streams, respectively.

2. We link semantics and implementation through semantics-directed compilation using partial evaluation [5, 17]. In particular, binding-time analysis guides us to extract templates from the specialized interpreters. These templates are similar to Proebsting's, and through partial evaluation, they give rise to similar flow-chart programs, demonstrating that templates are not just a good idea—they are intrinsic to the semantics of Icon and can be provably derived.

The rest of the paper is structured as follows: In Section 3.2 we first describe syntax and monadic semantics of a small subset of Icon; we then instantiate the semantics with various monads, relate the resulting semantics to each other, and present an equivalence proof for two of them. In Section 3.3 we describe semantics-directed compilation for a goal-directed language. Section 3.4 concludes.

3.2 Semantics of a Subset of Icon

An intuitive explanation of goal-directed evaluation can be given in terms of lists and list-manipulating functions. Consequently, after introducing the subset of Icon treated in this paper, we define a monadic semantics in terms of the list monad. We then show that also a stream monad and two different continuation monads can be used, and we give an example of how to prove equivalence of the resulting monads using a monad morphism.

3.2.1 A subset of the Icon programming language

We consider the following subset of Icon:

$$E ::= i \mid E_1 \texttt{ + } E_2 \mid E_1 \texttt{ to } E_2 \mid E_1 \texttt{ <= } E_2 \mid \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3$$

Intuitively, an Icon term either fails or succeeds with a value. If it succeeds, then subsequently it can be resumed, in which case it will again either succeed or fail. This process ends when the expression fails. Informally, i succeeds with the value i; E_1 `+` E_2 succeeds with the sum of the sub-expressions; E_1 `to` E_2 (called a *generator*) succeeds with the value of E_1 and each subsequent resumption yields the rest of the integers up to the value of E_2, at which point it fails; E_1 `<=` E_2 succeeds with the value of E_2 if it is larger than the value E_1, otherwise it fails; `if` E_1 `then` E_2 `else` E_3 produces the results of E_2 if E_1 succeeds, otherwise it produces the results of E_3.

Generators can be nested. For example, the Icon term `4 to (5 to 7)` generates the result of the expressions `4 to 5`, `4 to 6`, and `4 to 7` and concatenates the results.

In a functional language such as Scheme, ML or Haskell, we can achieve the effect of Icon terms using the functions `map` and `concat`. For example, if we define

```
fun to i j = if i<=j then i::(to (i+1) j) else nil
```

in ML, then evaluating `concat (map (to 4) (to 5 7))` yields `[4, 5, 4, 5, 6, 4, 5, 6, 7]` which is the list of the integers produced by the Icon term `4 to (5 to 7)`.

3.2.2 Monads and semantics

Computational monads were introduced as a tool for structuring denotational semantics [21]. The basic idea is to parameterize a semantics over a monad; many language extensions, such as adding a store or exceptions, can then be carried out by simply instantiating the semantics with a suitable monad. Further, correspondence proofs between semantics arising from instantiation with different monads can be conducted in a modular way, using the concept of a monad morphism [28].

Monads can also be used to structure functional programs [29]. In terms of programming languages, a monad M is described by a unary type constructor M and three operations $unit_{\mathsf{M}}$, map_{M} and $join_{\mathsf{M}}$ with types as displayed in Figure 3.1. For these operations, the so-called monad laws have to hold.

In Section 3.2.4 we give a denotational semantics of the goal-directed language described in Section 3.2.1. Anticipating semantics-directed compilation by partial evaluation, we describe the semantics in terms of ML, in effect defining an interpreter. The semantics $[\![\cdot]\!]_{\mathsf{M}} : Exp \to int\,\mathsf{M}$ is parameterized over a monad M, where $\alpha\,\mathsf{M}$ represents a sequence of values of type α.

3.2.3 A monad of sequences

In order to handle sequences, some structure is needed in addition to the three generic monad operations displayed in Figure 3.1. We add three operations:

$$\begin{array}{rcl} empty_{\mathsf{M}} & : & \alpha\,\mathsf{M} \\ if\_empty_{\mathsf{M}} & : & \alpha\,\mathsf{M} \to \beta\,\mathsf{M} \to \beta\,\mathsf{M} \to \beta\,\mathsf{M} \\ append_{\mathsf{M}} & : & \alpha\,\mathsf{M} \to \alpha\,\mathsf{M} \to \alpha\,\mathsf{M} \end{array}$$

Here, $empty_{\mathsf{M}}$ stands for the empty sequence; if_empty_{M} is a discriminator function that, given a sequence and two additional inputs, returns the first input if the sequence is empty, and returns the second input otherwise; $append_{\mathsf{M}}$ appends two sequences.

A straightforward instance of a monad of sequences is the list monad L, which is displayed in Figure 3.2; for lists, "join" is sometimes also called "flatten" or, in ML, "concat".

3.2.4 A monadic semantics

A monadic semantics of the goal-directed language described in Section 3.2.1. is given in Figure 3.3. We explain the semantics in terms of the list monad.

A literal i is interpreted as an expression that yields exactly one result; consequently, i is mapped into the singleton list $[i]$ using *unit*. The semantics of `to`, `+` and `<=` are given in terms of *bind2* and a function of type $int \to int \to int\,\mathsf{list}$. The type of function $bind2_{\mathsf{L}}$ is

$$(\alpha \to \beta \to \gamma\,\mathsf{list}) \to \alpha\,\mathsf{list} \to \beta\,\mathsf{list} \to \gamma\,\mathsf{list},$$

i.e., it takes two lists containing values of type α and β, and a function mapping $\alpha \times \beta$ into a list of values of type γ. The effect of the definition of $bind2_{\mathsf{L}}\ f\ xs\ ys$ is (1) to map $f\,x$ over ys for each x in xs and (2) to flatten the resulting list of lists. Both steps can be found in the example at the end of Section 3.2.1 of how the effect of goal-directed evaluation can be achieved in ML using lists.

$$
\begin{array}{lcl}
unit_{\mathsf{M}} & : & \alpha \to \alpha\,\mathsf{M} \\
map_{\mathsf{M}} & : & (\alpha \to \beta) \to \alpha\,\mathsf{M} \to \beta\,\mathsf{M} \\
join_{\mathsf{M}} & : & (\alpha\,\mathsf{M})\,\mathsf{M} \to \alpha\,\mathsf{M}
\end{array}
$$

Figure 3.1: Monad operators and their types

Standard monad operations:

$$
\begin{array}{lcl}
unit_{\mathsf{L}}\ x & = & [x] \\
map_{\mathsf{L}}\ f\ [\,] & = & [\,] \\
map_{\mathsf{L}}\ f\ (x :: xs) & = & (f\ x) :: (map_{\mathsf{L}}\ f\ xs) \\
join_{\mathsf{L}}\ [\,] & = & [\,] \\
join_{\mathsf{L}}\ (l :: ls) & = & l \mathbin{@} (join_{\mathsf{L}}\ ls)
\end{array}
$$

Special operations for sequences:

$$
\begin{array}{lcl}
empty_{\mathsf{L}} & = & [\,] \\
if\_empty_{\mathsf{L}}\ [\,]\ ys\ zs & = & ys \\
if\_empty_{\mathsf{L}}\ (x :: xs)\ ys\ zs & = & zs \\
append_{\mathsf{L}}\ xs\ ys & = & xs \mathbin{@} ys
\end{array}
$$

Figure 3.2: The list monad

$$
\begin{array}{rcl}
[\![\cdot]\!]_{\mathsf{M}} & : & Exp \to int\,\mathsf{M} \\
\\
[\![i]\!]_{\mathsf{M}} & = & unit_{\mathsf{M}}\ i \\
[\![E_1\ \texttt{to}\ E_2]\!]_{\mathsf{M}} & = & bind2_{\mathsf{M}}\ (\lambda xy.to_{\mathsf{M}}\ x\ y)\ [\![E_1]\!]_{\mathsf{M}}\ [\![E_2]\!]_{\mathsf{M}} \\
[\![E_1\ \texttt{+}\ E_2]\!]_{\mathsf{M}} & = & bind2_{\mathsf{M}}\ (\lambda xy.unit_{\mathsf{M}}\ (x+y))\ [\![E_1]\!]_{\mathsf{M}}\ [\![E_2]\!]_{\mathsf{M}} \\
[\![E_1\ \texttt{<=}\ E_2]\!]_{\mathsf{M}} & = & bind2_{\mathsf{M}}\ (\lambda xy.leq_{\mathsf{M}}\ x\ y)\ [\![E_1]\!]_{\mathsf{M}}\ [\![E_2]\!]_{\mathsf{M}} \\
[\![\texttt{if}\ E_0\ \texttt{then}\ E_1 & & \\
\quad \texttt{else}\ E_2]\!]_{\mathsf{M}} & = & if\_empty_{\mathsf{M}}\ [\![E_0]\!]_{\mathsf{M}}\ [\![E_1]\!]_{\mathsf{M}}\ [\![E_2]\!]_{\mathsf{M}}
\end{array}
$$

where

$$
\begin{array}{rcl}
bind2_{\mathsf{M}}\ f\ xs\ ys & = & join_{\mathsf{M}}\ (map_{\mathsf{M}}\ (\lambda x.join_{\mathsf{M}}\ (map_{\mathsf{M}}\ (f\ x)\ ys))\ xs) \\
leq_{\mathsf{M}}\ i\ j & = & \textbf{if}\ i \le j\ \textbf{then}\ unit_{\mathsf{M}}\ j\ \textbf{else}\ empty_{\mathsf{M}} \\
to_{\mathsf{M}}\ i\ j & = & \textbf{if}\ i > j\ \textbf{then}\ empty_{\mathsf{M}} \\
 & & \quad \textbf{else}\ append_{\mathsf{M}}\ (unit_{\mathsf{M}}\ i)\ (to_{\mathsf{M}}\ (i+1)\ j)
\end{array}
$$

Figure 3.3: Monadic semantics for a subset of Icon

Standard monad operations:

$$\begin{array}{lcl} unit_{\mathsf{C}}\ x & = & \lambda k.k\,x \\ map_{\mathsf{C}}\ f\ xs & = & \lambda k.xs\,(\lambda x.k\,(f\,x)) \\ join_{\mathsf{C}}\ ls & = & \lambda k.ls\,(\lambda x.x\,k) \end{array}$$

Special operations for sequences:

$$\begin{array}{lcl} empty_{\mathsf{C}} & = & \lambda k.\lambda l.l \\ if\_empty_{\mathsf{C}}\ xs\ ys\ zs & = & \lambda k.\lambda l.xs\,(\lambda\_.\lambda\_.ys\,k\,l)\,(zs\,k\,l) \\ append_{\mathsf{C}}\ xs\ ys & = & \lambda k.(xs\,k)\circ(ys\,k) \end{array}$$

Figure 3.4: The continuation monad

3.2.5 A spectrum of semantics

In the following, we describe four possible instantiations of the semantics given in Figure 3.3. Because a semantics corresponds directly to an interpreter, we thus create four different interpreters.

A list-based interpreter

Instantiating the semantics with the list monad from Figure 3.2 yields a list-based interpreter. In an eager language such as ML, a list-based interpreter always computes all results. Such behavior may not be desirable in a situation where only the first result is of interest (or, for that matter, whether there exists a result): Consider for example the conditional, which examines whether a given expression yields at least one result or fails. An alternative is to use laziness.

A stream-based interpreter

Implementing the list monad from Figure 3.2 in a lazy language results in a monad of (finite) lazy lists; the corresponding interpreter generates one result at a time. In an eager language, this effect can be achieved by explicitly implementing a data type of streams, i.e., finite lists built lazily: a thunk is used to delay computation.

$$\alpha\,\mathsf{stream} \;\equiv\; \mathsf{End} \mid \mathsf{More\ of}\ (\alpha \times (\mathbf{1} \to \alpha\,\mathsf{stream}))$$

The definition of the corresponding monad operations is straightforward.

A continuation-based interpreter

Gudeman [13] gives a continuation-based semantics of a goal-directed language. We can derive this semantics by instantiating our monadic semantics with the

continuation monad C as defined in Figure 3.4. The type-constructor $\alpha\,\mathsf{C}$ of the continuation monad is defined as $(\alpha \to R) \to R$, where R is called the *answer type* of the continuation.

A conceptual link between the list monad and the continuation monad with answer type $\beta\,\mathsf{list} \to \beta\,\mathsf{list}$ can be made through a Church encoding [4] of the higher-order representation of lists proposed by Hughes [16]. Hughes observed that when constructing the partially applied concatenation function $\lambda ys.xs \mathbin{@} ys$ rather than the list xs, lists can be appended in constant time. In the resulting representation, the empty list corresponds to the function that appends no elements, i.e., the identity, whereas the function that appends a single element is represented by a partially applied cons function:

$$\begin{array}{rcl} \mathit{nil} & = & \lambda ys.ys \\ \mathit{cons}\,x & = & \lambda ys.x :: ys \end{array}$$

Church-encoding a data types means abstracting over selector functions, in this case " :: ":

$$\begin{array}{rcl} \mathit{nil} & = & \lambda s_c.\lambda ys.ys \\ \mathit{cons}\,x & = & \lambda s_c.\lambda ys.s_c\,x\,ys \end{array}$$

The resulting representation of lists can be typed as

$$(\alpha \to \beta \to \beta) \to \beta \to \beta,$$

which indeed corresponds to $\alpha\,\mathsf{C}$ with answer type $\beta \to \beta$. Notice that *nil* and *cons* for this list representation yield $\mathit{empty}_{\mathsf{C}}$ and $\mathit{unit}_{\mathsf{C}}$, respectively. Similarly, the remaining monad operations correspond to the usual list operations.

Figure 3.5 displays the definition of $[\![\cdot]\!]_{\mathsf{C}}$ where all monad operations have been inlined and the resulting expressions β-reduced.

$$\begin{array}{rcl} [\![\cdot]\!]_{\mathsf{C}} & : & \mathit{Exp} \to (\mathit{int} \to \beta \to \beta) \to \beta \to \beta \\ \\ [\![i]\!]_{\mathsf{C}} & = & \lambda k.k\,i \\ [\![E_1\ \texttt{to}\ E_2]\!]_{\mathsf{C}} & = & \lambda k.[\![E_1]\!]_{\mathsf{C}}\,(\lambda i.[\![E_2]\!]_{\mathsf{C}}\,(\lambda j.\mathit{to}_{\mathsf{C}}\ i\ j\ k)) \\ [\![E_1\ \texttt{+}\ E_2]\!]_{\mathsf{C}} & = & \lambda k.[\![E_1]\!]_{\mathsf{C}}\,(\lambda i.[\![E_2]\!]_{\mathsf{C}}\,(\lambda j.k\,(i+j))) \\ [\![E_1\ \texttt{<=}\ E_2]\!]_{\mathsf{C}} & = & \lambda k.[\![E_1]\!]_{\mathsf{C}}\,(\lambda i.[\![E_2]\!]_{\mathsf{C}}\,(\lambda j.\mathit{leq}_{\mathsf{C}}\ i\ j\ k)) \\ [\![\texttt{if}\ E_0\ \texttt{then}\ E_1 & & \\ \quad \texttt{else}\ E_2]\!]_{\mathsf{C}_2} & = & \lambda k.\lambda l.[\![E_0]\!]_{\mathsf{C}_2}\,(\lambda\_.\lambda\_.[\![E_1]\!]_{\mathsf{C}_2}\,k\,l)\,([\![E_2]\!]_{\mathsf{C}_2}\,k\,l) \end{array}$$

where

$$\begin{array}{rcl} \mathit{leq}_{\mathsf{C}}\ i\ j & = & \lambda k.\mathbf{if}\ i \le j\ \mathbf{then}\ (k\ j)\ \mathbf{else}\ (\lambda l.l) \\ \mathit{to}_{\mathsf{C}}\ i\ j & = & \lambda k.\mathbf{if}\ i > j\ \mathbf{then}\ (\lambda l.l) \\ & & \qquad \mathbf{else}\ (k\ i) \circ (\mathit{to}_{\mathsf{C}}\ (i+1)\ j\ k) \end{array}$$

Figure 3.5: A continuation semantics

An interpreter with explicit success and failure continuations

A tail-recursive implementation of a continuation-based interpreter for Icon uses explicit success and failure continuations. The result of interpreting an Icon expression then has type

$$(int \to (\mathbf{1} \to \alpha) \to \alpha) \to (\mathbf{1} \to \alpha) \to \alpha,$$

where the first argument is the success continuation and the second argument the failure continuation. Note that the success continuation takes a failure continuation as a second argument. This failure continuation determines the resumption behavior of the Icon term: the success continuation may later on apply its failure continuation to generate more results. The corresponding continuation monad C_2 has the same standard monad operations as the continuation monad displayed in Figure 3.4, and the sequence operations

$$\begin{array}{lcl} \mathit{empty}_{\mathsf{C}_2} & = & \lambda k.\lambda f.f\,() \\ \mathit{if\_empty}_{\mathsf{C}_2}\ xs\ ys\ zs & = & \lambda k.\lambda f.xs\,(\lambda\_.\lambda\_.zs\,k\,f)\,(\lambda().ys\,k\,f) \\ \mathit{append}_{\mathsf{C}_2}\ xs\ ys & = & \lambda k.\lambda f.(xs\,k)(\lambda().ys\,k\,f) \end{array}$$

Just as the continuation monad from Figure 3.4 can be conceptually linked to the list monad, the present continuation monad can be linked to the stream monad by a Church encoding of the data type of streams:

$$\begin{array}{lcl} \mathit{end} & = & \lambda s_m.\lambda s_e.s_e() \\ \mathit{more}\,x\,xs & = & \lambda s_m.\lambda s_e.s_m\,x\,xs \end{array}$$

The fact that the second component in a stream is a thunk suggests one to give the selector function s_m the type $int \to (\mathbf{1} \to \alpha) \to \beta$; the resulting type for *end* and $\mathit{more}\,x\,xs$ is then

$$(int \to (\mathbf{1} \to \alpha) \to \beta) \to (\mathbf{1} \to \beta) \to \beta.$$

Choosing α as the result type of the selector functions yields the type of a continuation monad with answer type $(\mathbf{1} \to \alpha) \to \alpha$.

The interpreter defined by the semantics $[\![\cdot]\!]_{\mathsf{C}_2}$ is the starting point of the semantics-directed compilation described in Section 3.3. Figure 3.6 displays the definition of $[\![\cdot]\!]_{\mathsf{C}_2}$ where all monad operations have been inlined and the resulting expressions β-reduced. Because the basic monad operations of C_2 are the same as those of C, the semantics based on C_2 and C only differ in the definitions of *leq*, *to*, and in how `if` is handled.

3.2.6 Correctness

So far, we have related the various semantics presented in Section 3.2.5 only conceptually. Because the four different interpreters presented in Section 3.2.5 were created by instantiating one parameterized semantics with different monads, a *formal* correspondence proof can be conducted in a modular way building on the concept of a monad morphism [28].

$$
\begin{array}{rcl}
[\![\cdot]\!]_{\mathsf{C}_2} & : & \mathit{Exp} \to (\mathit{int} \to (\mathbf{1} \to \alpha) \to \alpha) \to (\mathbf{1} \to \alpha) \to \alpha \\
\\
[\![i]\!]_{\mathsf{C}_2} & = & \lambda k.k\,i \\
[\![E_1\ \texttt{to}\ E_2]\!]_{\mathsf{C}_2} & = & \lambda k.[\![E_1]\!]_{\mathsf{C}_2}\,(\lambda i.[\![E_2]\!]_{\mathsf{C}_2}\,(\lambda j.\mathit{to}_{\mathsf{C}_2}\ i\ j\ k)) \\
[\![E_1\ \texttt{+}\ E_2]\!]_{\mathsf{C}_2} & = & \lambda k.[\![E_1]\!]_{\mathsf{C}_2}\,(\lambda i.[\![E_2]\!]_{\mathsf{C}_2}\,(\lambda j.k\,(i+j))) \\
[\![E_1\ \texttt{<=}\ E_2]\!]_{\mathsf{C}_2} & = & \lambda k.[\![E_1]\!]_{\mathsf{C}_2}\,(\lambda i.[\![E_2]\!]_{\mathsf{C}_2}\,(\lambda j.\mathit{leq}_{\mathsf{C}_2}\ i\ j\ k)) \\
[\![\texttt{if}\ E_0\ \texttt{then}\ E_1 & & \\
\quad \texttt{else}\ E_2]\!]_{\mathsf{C}_2} & = & \lambda k.\lambda f.[\![E_0]\!]_{\mathsf{C}_2}\,(\lambda\_.\lambda\_.[\![E_1]\!]_{\mathsf{C}_2}\,k\,f)\,(\lambda().[\![E_2]\!]_{\mathsf{C}_2}\,k\,f)
\end{array}
$$

where

$$
\begin{array}{rcl}
\mathit{leq}_{\mathsf{C}_2}\ i\ j & = & \lambda k.\lambda f.\mathbf{if}\ i \le j\ \mathbf{then}\ k\ j\ f\ \mathbf{else}\ f\,() \\
\mathit{to}_{\mathsf{C}_2}\ i\ j & = & \lambda k.\lambda f.\mathbf{if}\ i > j\ \mathbf{then}\ f\,() \\
 & & \quad \mathbf{else}\ (k\,i)\,(\lambda().\mathit{to}_{\mathsf{C}_2}\ (i+1)\ j\,k\,f)
\end{array}
$$

Figure 3.6: A semantics with success and failure continuations

Definition 3.1 (Monad morphism) *If* M *and* N *are two monads, then* $h : \alpha\,\mathsf{M} \to \alpha\,\mathsf{N}$ *is a* monad morphism *if it preserves the monad operations*[1]*, i.e.,*

$$
\begin{array}{rcl}
h \circ \mathit{unit}_{\mathsf{M}} & = & \mathit{unit}_{\mathsf{N}} \\
h \circ \mathit{map}_{\mathsf{M}}\,f & = & \mathit{map}_{\mathsf{N}}\,f \circ h \\
h \circ \mathit{join}_{\mathsf{M}} & = & \mathit{join}_{\mathsf{N}} \circ h \circ \mathit{map}_{\mathsf{M}}\,h \\
h\ \mathit{empty}_{\mathsf{M}} & = & \mathit{empty}_{\mathsf{N}} \\
h \circ \mathit{if\_empty}_{\mathsf{M}} & = & \lambda xs.\lambda ys.\lambda zs.\mathit{if\_empty}_{\mathsf{N}}(h\,xs)(h\,ys)(h\,zs) \\
h \circ \mathit{append}_{\mathsf{M}} & = & \lambda xs.\lambda ys.\mathit{append}_{\mathsf{N}}(h\,xs)(h\,ys)
\end{array}
$$

The following lemma shows that the semantics resulting from two different monad instantiations can be related by defining a monad morphism between the two sequence monads in question.

Lemma 3.2 *Let* M *and* N *be monads of sequences as specified in Section 3.2.3. If* h *is a monad morphism from* M *to* N*, then* $(h\,[\![E]\!]_{\mathsf{M}}) = [\![E]\!]_{\mathsf{N}}$ *for every Icon expression* E*.*

Proof: By induction over the structure of E. A lemma to the effect that $h\ (\mathit{to}_{\mathsf{M}}\ i\ j) = \mathit{to}_{\mathsf{N}}\,i\,j$ is shown by induction over $i - j$ for $i \ge j$. □

We use Lemma 3.2 to show that the list-based interpreter from Section 3.2.5 and the continuation-based interpreter from Section 3.2.5 always yield comparable results:

[1] We strengthen the definition of a monad morphism somewhat by considering a *sequence-preserving* monomorphism that also preserves the monad operations specific to the monad of sequences.

Proposition 3.3 *Let* $show : \alpha\,\mathsf{C} \to \alpha\,\mathsf{L}$ *be defined as*

$$show\ f\ =\ f\ (\lambda x.\lambda xs.append_{\mathsf{L}}\ (unit_{\mathsf{L}}\ x)\ xs)\ empty_{\mathsf{L}}.$$

Then $(show\ [\![E]\!]_{\mathsf{C}}) = [\![E]\!]_{\mathsf{L}}$ *for all Icon expressions* E.

Proof: We show that (1) $h : \alpha\,\mathsf{L} \to \alpha\,\mathsf{C}$, which is defined as

$$\begin{array}{rcl} h\ [\,] & = & empty_{\mathsf{C}} \\ h\ (x :: xs) & = & append_{\mathsf{C}}\ (unit_{\mathsf{C}}\ x)\ (h\ xs) \end{array}$$

is a monad morphism from L to C, and (2) the function $(show \circ h)$ is the identity function on lists. The proposition then follows immediately with Lemma 3.2. □

3.2.7 Conclusion

Taking an intuitive list-based semantics for a subset of Icon as our starting point, we have defined a stream-based semantics and two continuation semantics. Because our inital semantics is defined as the instantiation of a monadic semantics with a list monad, the other semantics can be defined through a stream monad and two different continuation monads, respectively. The modularity of the monadic semantics allows us to relate the semantics to each other by relating the corresponding monads, both conceptually and formally. To the best of our knowledge, the conceptual link between list-based monads and continuation monads via Church encoding has not been observed before.

It is known that continuations can be compiled into efficient code relatively easily [1, 14]; in the following section we show that partial evaluation is sufficient to generate efficient code from the the continuation semantics derived in Section 3.2.5.

3.3 Semantics-Directed Compilation

The goal of partial evaluation is to specialize a source program $p : S \times D \to R$ of two arguments to a fixed "static" argument $s : S$. The result is a residual program $p_s : D \to R$ that must yield the same result when applied to a "dynamic" argument d as the original program applied to both the static and the dynamic arguments, i.e., $[\![p_s(d)]\!] = [\![p(s, d)]\!]$.

Our interest in partial evaluation is due to its use in semantics-directed compilation: when the source program p is an interpreter and the static argument s is a term in the domain of p then p_s is a compiled version of s represented in the implementation language of p. It is often possible to implement an interpreter in a functional language based on the denotational semantics.

Our starting point is a functional interpreter implementing the denotational semantics in Figure 3.6. The source language of the interpreter is shown in Figure 3.7. In Section 3.3.1 we present the Icon interpreter written in ML.

In Section 3.3.1, 3.3.2, and 3.3.3 we use type-directed partial evaluation to specialize this interpreter to Icon terms yielding ML code, C code, and OCaml byte code as output. Other partial-evaluation techniques could be applied to yield essentially the same results.

```
structure Icon = struct
  datatype icon = LIT  of int
                | TO   of icon * icon
                | PLUS of icon * icon
                | LEQ  of icon * icon
                | IF   of icon * icon * icon
end
```

Figure 3.7: The abstract syntax of Icon terms

3.3.1 Type-directed partial evaluation

We have used type-directed partial evaluation to compile Icon programs into ML. This is a standard exercise in semantics-directed compilation using type-directed partial evaluation [9].

Type-directed partial evaluation is an approach to off-line specialization of higher-order programs [8]. It uses a normalization function to map the (value of the) trivially specialized program $\lambda d.p(s, d)$ into the (text of the) target program p_s.

The input to type-directed partial evaluation is a binding-time separated program in which static and dynamic primitives are separated. When implemented in ML, the source program is conveniently wrapped in a functor parameterized over a structure of dynamic primitives. The functor can be instantiated with evaluating primitives (for running the source program) and with residualizing primitives (for specializing the source program).

Specializing Icon terms using type-directed partial evaluation

In our case the dynamic primitives operations are addition (`add`), integer comparison (`leq`), a fixed-point operator (`fix`), a conditional functional (`cond`), and a quoting function (`qint`) lifting static integers into the dynamic domain. The signature of primitives is shown in Figure 3.8. For the residualizing primitives we let the partial evaluator produce functions that generate ML programs with meaningful variable names [8].

The parameterized interpreter is shown in Figure 3.9. The main function `eval` takes an Icon term and two continuations, `k` : `tint` → (`tunit` → `res`) → `res` and `f` : `tunit` → `res`, and yields a result of type `res`. We intend to specialize the interpreter to a static Icon term and keeping the continuation parameters `k` and `f` dynamic. Consequently, residual programs are parameterized over two

continuations. (If the continuations were also considered static then the residual programs would simply be the list of the generated integers.)

```
signature PRIMITIVES = sig
  type tunit
  type tint
  type tbool
  type res

  val qint : int -> tint
  val add  : tint * tint -> tint
  val leq  : tint * tint -> tbool
  val cond : tbool * (tunit -> res) * (tunit -> res) -> res
  val fix  : ((tint -> res) -> tint -> res) -> tint -> res
end
```

Figure 3.8: Signature of primitive operations

The output of type-directed partial evaluation is the text of the residual program. The residual program is in long beta-eta normal form, that is, it does not contain any beta redexes and it is fully eta-expanded with respect to its type.

Example 3.4 *The following is the result of specializing the interpreter with respect to the Icon term* `10 + (4 to 7)`*.*

```
fn k => fn f =>
   fix (fn loop0 =>
           fn i0 =>
              cond (leq (i0, qint 7),
                    fn () => k (add (qint 10, i0))
                                (fn () => loop0 (add (i0,
                                                      qint 1))),
                    fn () => f ()))
       (qint 4)
```

Avoiding code duplication

The result of specializing the interpreter in Figure 3.9 may be exponentially large. This is due to the continuation parameter `k` being duplicated in the clause for `IF`. For example, specializing the interpreter to the Icon term `100 + (if 1 < 2 then 3 else 4)` yields the following residual program in which the context `add(100, ·)` occurs twice.

```
functor MakeInterp(P : PRIMITIVES) = struct
  fun loop (i, j) k f =
      P.fix
        (fn walk =>
            fn i =>
               P.cond (P.leq (i, j),
                       fn _ =>
                          k i (fn _ =>
                                  walk (P.add (i, P.qint 1))),
                       f))
        i

  fun select (i, j) k f =
      P.cond (P.leq (i, j), fn _ => k j f, f)

  fun sum (i, j) k = k (P.add (i, j))

  fun eval (LIT i)          k = k (P.qint i)
    | eval (TO(e1, e2))     k =
      eval e1 (fn i => eval e2 (fn j => loop (i, j) k))
    | eval (PLUS(e1, e2))   k =
      eval e1 (fn i => eval e2 (fn j => sum (i, j) k))
    | eval (LEQ(e1, e2))    k =
      eval e1 (fn i => eval e2 (fn j => select (i, j) k))
    | eval (IF(e1, e2, e3)) k =
      fn f =>
         eval e1
              (fn _ => fn _ => eval e2 k f)
              (fn _ => eval e3 k f)
end
```

Figure 3.9: Parameterized interpreter

```
fn k => fn f =>
   cond (leq (qint 1, qint 2),
         fn () => k (add (qint 100, qint 3)) (fn () => f ()),
         fn () => k (add (qint 100, qint 4)) (fn () => f ()))
```

Code duplication is a well-known problem in partial evaluation [17]. The equally well-known solution is to bind the continuation in the residual program, just before it is used. We introduce a new primitive `save` of two arguments, `k` and `g`, which applies `g` to two "copies" of the continuation `k`.

```
signature PRIMITIVES = sig
  ...
  type succ = tint -> (tunit -> res) -> res
  val save : succ -> (succ * succ -> res) -> res
end
```

The final clause of the interpreter is modified to save the continuation parameter before it proceeds, as follows.

```
fun eval (LIT i)          k = k (P.qint i)
    ...
  | eval (IF(e1, e2, e3)) k =
    fn f =>
       save k
        (fn (k0, k1) => eval e1
                             (fn _ => fn _ => eval e2 k0 f)
                             (fn _ => eval e3 k1 f))
```

Specializing this new interpreter to the Icon term from above yields the following residual program in which the context `add(100, ·)` occurs only once.

```
fn k => fn f =>
   save (fn v0 =>
            fn resume0 =>
               k (add (qint 100, v0)) (fn () => resume0 ()))
        (fn (k0_0, k1_0) =>
            cond (leq (qint 1, qint 2),
                  fn () => k0_0 (qint 3) (fn () => f ()),
                  fn () => k1_0 (qint 4) (fn () => f ())))
```

Two copies of continuation parameter `k` are bound to `k0_0` and `k1_0` before the continuation is used (twice, in the body of the second lambda). In order just to prevent code duplication, passing one "copy" of the continuation parameter is actually enough. But the translation into C introduced in Section 3.3.2 uses the two differently named variables, in this case `k0_0` and `k1_0`, to determine the `IF`-branch inside which a continuation is applied.

3.3.2 Generating C programs

Residual programs are not only in long beta-eta normal form. Their type

$$(\texttt{tint} \to (\texttt{tunit} \to \texttt{res}) \to \texttt{res}) \to (\texttt{tunit} \to \texttt{res}) \to \texttt{res}$$

imposes further restrictions: A residual program must take two arguments, a success continuation $\texttt{k} : \texttt{tint} \to (\texttt{tunit} \to \texttt{res}) \to \texttt{res}$ and a failure continuation $\texttt{f} : \texttt{tunit} \to \texttt{res}$, and it must produce a value of type `res`. When we also consider the types of the primitives that may occur in residual programs we see that values of type `res` can only be a result of

- applying the success continuation `k` to an integer n and function of type `tunit` → `res`;
- applying the failure continuation `f`;
- applying `cond` to a boolean and two functions of type `tunit` → `res`;
- applying `fix` to a function of two arguments, $\texttt{loop}_n$: `tint` → `res` and $\texttt{i}_n$: `tint`, and an integer;
- (inside a function passed to `fix`) applying the function $\texttt{loop}_n$ to an integer;
- applying `save` to two arguments, the first being a function of two arguments, $\texttt{v}_n$: `tint` and $\texttt{resume}_n$: `tunit` → `res`, and the second being a function of a pair of arguments, $\texttt{k}_n^0$ and $\texttt{k}_n^1$, each of type `tint` → (`tunit` → `res`) → `res`;
- (inside the first function passed to `save`) applying the function $\texttt{resume}_n$; or
- (inside the second function passed to `save`) applying one of the functions $\texttt{k}_n^0$ or $\texttt{k}_n^1$ to an integer and a function of type `tunit` → `res`.

A similar analysis applies to values of type `tint`: they can only arise from evaluating an integer n, a variable $\texttt{i}_n$, or a variable $\texttt{v}_n$ or from applying `add` to two argument of type `tint`. As a result, we observe that the residual programs of specializing the Icon interpreter using type-directed partial evaluation are restricted to the grammar in Figure 3.10. (The restriction that the variables $\texttt{loop}_n$, $\texttt{i}_n$, $\texttt{v}_n$, and $\texttt{resume}_n$ each must occur inside a function that binds them cannot be expressed using a context-free grammar. This is not a problem for our development.) We have expressed the grammar as an ML datatype and used this datatype to represent the output from type-directed partial evaluation. Thus, we have essentially used the type system of ML as a theorem prover to show the following lemma.

Lemma 3.5 *The residual program generated from applying type-directed partial evaluation to the interpreter in Figure 3.9 can be generated by the grammar in Figure 3.10.*

The idea of generating grammars for residual programs has been studied by, e.g., Malmkjær [20] and is used in the run-time specializer Tempo to generate code templates [6].

The simple structure of output programs allows them to be viewed as programs of a flow-chart language. We choose C as a concrete example of such a language. Figure 3.11 and 3.12 show the translation from residual programs to C programs.

The translation replaces function calls with jumps. Except for the call to $\texttt{resume}_n$ (which only occurs as the result of compiling if-statements), the name of a function uniquely determines the corresponding label to jump to. Jumps to

$$
\begin{array}{rcl}
I & ::= & \texttt{fn k => fn f => } S \\
S & ::= & \texttt{k } E \texttt{ (fn () => } S\texttt{)} \\
 & | & \texttt{f ()} \\
 & | & \texttt{cond (}E\texttt{, fn () => } S\texttt{, fn () => } S\texttt{)} \\
 & | & \texttt{fix (fn loop}_n \texttt{ => fn i}_n \texttt{ => } S\texttt{) } E \\
 & | & \texttt{loop}_n\ E \\
 & | & \texttt{save (fn v}_n \texttt{ => fn resume}_n \texttt{ => } S\texttt{) (fn (k}_n^0\texttt{, k}_n^1\texttt{) => } S\texttt{)} \\
 & | & \texttt{resume}_n\ \texttt{()} \\
 & | & \texttt{k}_n^i\ E\ \texttt{(fn () => } S\texttt{)}, \quad \text{where } i \in \{0,1\} \\
E & ::= & \texttt{qint } n \mid \texttt{i}_n \mid \texttt{v}_n \mid \texttt{add (}E\texttt{, } E\texttt{)} \mid \texttt{leq (}E\texttt{, } E\texttt{)}
\end{array}
$$

Figure 3.10: Grammar of residual programs

$\texttt{resume}_n$ can end up in two different places corresponding to the two copies of the continuation. We use a boolean variable $\texttt{gate}_n$ to distinguish between the two possible destinations. Calls to $\texttt{loop}_n$ and $\texttt{k}_n$ pass arguments. The names of the formal parameters are known ($\texttt{i}_n$ and $\texttt{v}_n$, respectively) and therefore arguments are passed by assigning the variable before the jump.

In each translation of a conditional a new label l must be generated. The entire translated term must be wrapped in a context that defines the labels `succ` and `fail` (corresponding to the initial continuations). The statements following the label `succ` are allowed to jump to `resume`. The translation in Figure 3.11 and 3.12 generates a C program that successively prints the produced integers one by one. A lemma to the effect that the translation from residual ML programs into C is semantics preserving would require giving semantics to C and to the subset of ML presented in Figure 3.10 and then showing equivalence.

Example 3.6 *Consider again the Icon term* `10 + (4 to 7)` *from Example 3.4. It is translated into the following C program.*

```
        i0 = 4;
loop0:  if (i0 <= 7) goto L0;
        goto fail;

L0:     value = 10 + i0;
        goto succ;

resume: i0 = i0 + 1;
        goto loop0;

succ:   printf("%d ", value);
        goto resume;

fail:   printf("\n");
        exit(0);
```

$$|\texttt{fn k => fn f => } S|_{\mathrm{I}} = \begin{cases} & |S|_{\mathrm{S}} \\ \texttt{succ:} & \texttt{printf("\%d ", value);} \\ & \texttt{goto resume;} \\ \texttt{fail:} & \texttt{printf("\textbackslash n");} \\ & \texttt{exit(0);} \end{cases}$$

$$|\texttt{k } E \texttt{ (fn () => } S\texttt{)}|_{\mathrm{S}} = \begin{cases} & \texttt{value = } |E|_{\mathrm{E}}\texttt{;} \\ & \texttt{goto succ;} \\ \texttt{resume:} & |S|_{\mathrm{S}} \end{cases}$$

$$|\texttt{f ()}|_{\mathrm{S}} = \{\texttt{goto fail;}$$

$$|\texttt{cond (}E\texttt{, fn () => }S\texttt{, fn () => }S'\texttt{)}|_{\mathrm{S}} = \begin{cases} & \texttt{if (}|E|_{\mathrm{E}}\texttt{) goto } l\texttt{;} \\ & |S'|_{\mathrm{S}} \\ l\texttt{:} & |S|_{\mathrm{S}} \end{cases}$$

$$|\texttt{fix (fn loop}_n \texttt{ => fn i}_n \texttt{ => } S\texttt{) } E|_{\mathrm{S}} = \begin{cases} & \texttt{i}_n \texttt{ = } |E|_{\mathrm{E}}\texttt{;} \\ \texttt{loop}_n\texttt{:} & |S|_{\mathrm{S}} \end{cases}$$

$$|\texttt{loop}_n \ E|_{\mathrm{S}} = \begin{cases} \texttt{i}_n \texttt{ = } |E|_{\mathrm{E}}\texttt{;} \\ \texttt{goto loop}_n\texttt{;} \end{cases}$$

$$\left|\begin{array}{l}\texttt{save (fn v}_n \texttt{ => fn resume}_n \texttt{ => } S\texttt{)} \\ \qquad\texttt{(fn (k}_n^0\texttt{, k}_n^1\texttt{) => } S'\texttt{)}\end{array}\right|_{\mathrm{S}} = \begin{cases} & |S'|_{\mathrm{S}} \\ \texttt{succ}_n\texttt{:} & |S|_{\mathrm{S}} \end{cases}$$

$$|\texttt{resume}_n \ ()|_{\mathrm{S}} = \begin{cases} \texttt{if (gate}_n\texttt{) goto resume}_n^1\texttt{;} \\ \texttt{goto resume}_n^0\texttt{;} \end{cases}$$

$$|\texttt{k}_n^i \ E \texttt{ (fn () => } S\texttt{)}|_{\mathrm{S}} = \begin{cases} & \texttt{gate}_n \texttt{ = } i\texttt{;} \\ & \texttt{v}_n \texttt{ = } |E|_{\mathrm{E}}\texttt{;} \\ & \texttt{goto succ}_n\texttt{;} \\ \texttt{resume}_n^i\texttt{:} & |S|_{\mathrm{S}} \end{cases}$$

Figure 3.11: Translating residual programs into C (Statements)

$$
\begin{array}{rcl}
|\texttt{qint } n|_{\mathrm{E}} & = & n \\
|\texttt{i}_n|_{\mathrm{E}} & = & \texttt{i}_n \\
|\texttt{v}_n|_{\mathrm{E}} & = & \texttt{v}_n \\
|\texttt{add (}E\texttt{, }E'\texttt{)}|_{\mathrm{E}} & = & |E|_{\mathrm{E}} \texttt{ + } |E'|_{\mathrm{E}} \\
|\texttt{leq (}E\texttt{, }E'\texttt{)}|_{\mathrm{E}} & = & |E|_{\mathrm{E}} \texttt{ <= } |E'|_{\mathrm{E}}
\end{array}
$$

Figure 3.12: Translating residual programs into C (Expressions)

The C target programs corresponds to the target programs of Proebsting's optimized template-based compiler [22]. In effect, we are automatically generating flow-chart programs from the denotation of an Icon term.

3.3.3 Generating byte code

In the previous two sections we have developed two compilers for Icon terms, one that generates ML programs and one that generates flow-chart programs. In this section we unify the two by composing the first compiler with an automatic run-time code generation system for OCaml [25] and by composing the second compiler with a hand-written compiler from flow charts into OCaml byte code.

Run-time code generation in OCaml

Run-time code generation for OCaml works by a deforested composition of traditional type-directed partial evaluation with a compiler into OCaml byte code. Deforestation is a standard improvement in run-time code generation [6, 19, 26]. As such, it removes the need to manipulate the text of residual programs at specialization time. As a result, instead of generating ML terms, run-time code generation allows type-directed partial evaluation to directly generate executable OCaml byte code.

Specializing the Icon interpreter from Figure 3.9 to the Icon term `10 + (4 to 7)` using run-time code generation yields a residual program of about 110 byte-code instructions in which functions are implemented as closures and calls are implemented as tail-calls. (Compiling the residual ML program using the OCaml compiler yields about 90 byte-code instructions.)

Compiling flow charts into OCaml byte code

We have modified the translation in Figure 3.11 and 3.12 to produce OCaml byte-code instructions instead of C programs. The result is an embedding of Icon into OCaml.

Using this compiler, `10 + (4 to 7)` yields 36 byte-code instructions in which functions are implemented as labelled blocks and calls are implemented as an assignment (if an argument is passed) followed by a jump. This style of target code was promoted by Steele in the first compiler for Scheme [27].

3.3.4 Conclusion

Translating the continuation-based denotational semantics into an interpreter written in ML and using type-directed partial evaluation enables a standard semantics-directed compilation from Icon terms into ML. A further compilation of residual programs into C yields flow-chart programs corresponding to those produced by Proebsting's Icon compiler [22].

3.4 Conclusions and Issues

Observing that the list monad provides the kind of backtracking embodied in Icon, we have specified a semantics of Icon that is parameterized by this monad. We have then considered alternative monads and proven that they also provide a fitting semantics for Icon. Inlining the continuation monad, in particular, yields Gudeman's continuation semantics [13].

Using partial evaluation, we have then specialized these interpreters with respect to Icon programs, thereby compiling these programs using the first Futamura projection. We used a combination of type-directed partial evaluation and code generation, either to ML, to C, or to OCaml byte code. Generating code for C, in particular, yields results similar to Proebsting's compiler [22].

Gudeman [13] shows that a continuation semantics can also deal with additional control structures and state; we do not expect any difficulties with scaling up the code-generation accordingly. The monad of lists, on the other hand, does not offer enough structure to deal, e.g., with state. It should be possible, however, to create a rich enough monad by combining the list monad with other monads such as the state monad [10, 18].

It is our observation that the traditional (in partial evaluation) generalization of the success continuation avoids the code duplication that Proebsting presents as problematic in his own compiler. We are also studying the results of defunctionalizing the continuations, à la Reynolds [24], to obtain stack-based specifications and the corresponding run-time architectures.

Acknowledgments

Thanks are due to the anonymous referees for comments and to Andrzej Filinski for discussions.

Bibliography

[1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, 1992.

[2] Lawrence Byrd. Understanding the control of Prolog programs. Technical Report 151, University of Edinburgh, 1980.

[3] Mats Carlsson. On implementing Prolog in functional programming. *New Generation Computing*, 2(4):347–359, 1984.

[4] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[5] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[6] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In Guy L. Steele, editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, Florida, January 1996. ACM Press.

[7] Ron K. Cytron, editor. *Proceedings of the ACM SIGPLAN'97 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 32, No 5, Las Vegas, Nevada, June 1997. ACM Press.

[8] Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag.

[9] Olivier Danvy and René Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. In Herbert Kuchen and Doaitse Swierstra, editors, *Eighth International Symposium on Programming Language Implementation and Logic Programming*, number 1140 in Lecture

Notes in Computer Science, pages 182–197, Aachen, Germany, September 1996. Springer-Verlag. Extended version available as the technical report BRICS-RS-96-13.

[10] Andrzej Filinski. Representing layered monads. In Alex Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, January 1999. ACM Press.

[11] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Prentice Hall, Inc., 1983.

[12] Ralph E. Griswold and Madge T. Griswold. *The Implementation of the Icon Programming Language*. Princeton University Press, 1986.

[13] David A. Gudeman. Denotational semantics of a goal-directed language. *ACM Transactions on Programming Languages and Systems*, 1992.

[14] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In Bernard Lang, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 25, No 6, pages 66–77, White Plains, New York, June 1990. ACM Press.

[15] Ralf Hinze. Prological features in a functional setting—axioms and implementations. In Masahiko Sato and Yoshihito Toyama, editors, *Third Fuji International Symposium on Functional and Logic Programming (FLOPS'98)*, pages 98–122, Kyoto, Japan, April 1998. World Scientific.

[16] John Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, 1986.

[17] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/pebook.html`.

[18] David J. King and Philip Wadler. Combining Monads. In John Launchbury and Patrick M. Sansom, editors, *Glasgow Workshop on Functional Programming*, Workshops in Computing, Ayr, Scotland, 1992. Springer, Berlin.

[19] Mark Leone and Peter Lee. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 31, No 5, pages 137–148. ACM Press, May 1996.

[20] Karoline Malmkjær. *Abstract Interpretation of Partial-Evaluation Algorithms*. PhD thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, March 1993.

[21] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[22] Todd A. Proebsting. Simple translation of goal-directed evaluation. In Cytron [7], pages 1–6.

[23] Todd A. Proebsting and Gregg M. Townsend. A new implementation of the Icon language. Technical Report 99-13, University of Arizona, Department of Computer Science, 1999.

[24] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

[25] Morten Rhiger. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, 2001. Forthcoming.

[26] Michael Sperber and Peter Thiemann. Two for the price of one: composing partial evaluation and compilation. In Cytron [7], pages 215–225.

[27] Guy L. Steele Jr. Compiler optimization based on viewing LAMBDA as RENAME + GOTO. In Patrick Henry Winston and Richard Henry Brown, editors, *Artificial Intelligence: An MIT Perspective*, volume 2. The MIT Press, 1979.

[28] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, December 1992.

[29] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 24–52. Springer-Verlag, 1995.

[30] Richard S. Wallace. An easy implementation of pil (PROLOG in LISP). *Association for Computing Machinery Special Interest Group on Artificial Intelligence. SIGART NEWSL.*, (85):29–32, July 1983.

Chapter 4

Partial Evaluation of Pattern Matching in Strings, revisited

Abstract

Specialization of a string matcher is a canonical example of partial evaluation. A naive implementation of a string matcher repeatedly matches a pattern against every substring of the data string; this operation should intuitively benefit from specializing the matcher with respect to the pattern. In practice, however, producing an efficient implementation by performing this specialization using standard partial-evaluation techniques requires non-trivial binding-time improvements. Starting with a naive matcher, we thus present a derivation of such a binding-time improved string matcher. We show that specialization with respect to a pattern yields a matcher with code size linear in the length of the pattern and a running time independent of the length of the pattern and linear in the length of the data string. We then consider several variants of matchers that specialize well, amongst them the first such matcher presented in the literature, and we demonstrate how variants can be derived from each other systematically.

4.1 Introduction

The Knuth-Morris-Pratt string-matching algorithm (KMP) [19] tests whether a pattern string p occurs in a data string d in time $O(|p| + |d|)$. The algorithm calculates a failure table from the pattern in time $O(|p|)$; using this failure table, the data string can be traversed without backtracking. Although the KMP can

be written in a few lines, it proves surprisingly difficult to comprehend. This property has made it a fruitful topic in the area of program transformation, from Knuth's original derivation onwards.

Partial evaluation is an automatic program transformation that specializes a program with respect to partial information about the input. Because a string matcher conceptually matches a single pattern against every possible starting position in the data string, string matching seems like a compelling target for partial evaluation. In effect, partial evaluation with respect to the pattern should have an effect comparable to calculating a failure table. Indeed, this "KMP-test" has become a popular benchmark for partial evaluators and related systems [28]. The systems that pass the KMP test have the ability to infer information about the (unknown) data string based on the form of enclosing conditional tests [8, 27, 28]. Such capability, however, goes beyond standard partial evaluation.

Another approach is to rewrite a naive string matcher to make it more amenable to standard partial evaluation by augmenting the implementation with static data that records the results of tests on the dynamic data: A standard partial evaluator, such as Mix [15], Schism [6] or Similix [5], can generate an efficient implementation from such a modified version. This insight, which inspired further investigations into the applicability of program-specialization systems to the string-matching problem, is due to Consel and Danvy [7]. Modifications to the source program to improve the result of partial evaluation are common in practical applications of partial evaluation, and are known as *binding-time improvements* [14, Chapter 12]. But, because in previous applications of this technique to string matching the naive matcher is modified in a single step [2, 7, 14, 28], it is neither obvious that the modifications preserve semantics, nor clear how such binding-time improvements can be achieved in a systematic way.

In this paper we present a stepwise derivation of a matcher; (1) we modify the naive matcher such that useful information is recorded, and (2) we apply standard binding-time improvements so as to increase the amount of static computation. In this derivation, only the first step is problem specific; we hope that this division illuminates how the approach can be applied to other pattern-matching problems and implementations. We prove that specializing our matcher with respect to a pattern string yields a residual program that has size linear in the length of the pattern and a pattern-independent running time linear in the length of the data string. Our first implementation only records the information that can be derived from the truth of enclosing conditional tests (*positive information*). The result of partial evaluation of this implementation runs in time linear in the length of the data string (independent of the pattern), but may perform some redundant tests. Therefore we modify the implementation to record information that can be derived from the falsity of enclosing conditional tests (*negative information*); the result of applying partial evaluation to this implementation never compares a character of the data string to the same pattern character more than once. Finally, we analyze how two published matchers (namely those of Consel and Danvy [7] and of Jones, Gomard, and

Sestoft [14]) can be derived from our matcher. In doing so, we explore a number of variations that specialize well with partial evaluators such as Similix.

The paper is structured as follows: Section 4.2 gives a short overview of the concepts of partial evaluation essential for this paper. In Section 4.3, a straightforward implementation of a string matcher is presented. Section 4.4 describes a derivation of a string matcher that records positive information, and proves that specializing this matcher yields a residual program of size linear in the length of the pattern and with a pattern-independent running time linear in the size of the data string. Section 4.5 does the same for a matcher that also records negative information, and shows that no redundant tests are performed by the specialized matcher. Section 4.6 discusses possible variations in the design of string matchers amenable for specialization, and shows how the matchers of Consel and Danvy and of Jones et al. can be derived from our implementation. Section 4.7 treats related work and Section 4.8 concludes.

4.2 Partial evaluation

Partial evaluation is an automatic program transformation that uses interprocedural constant propagation to specialize a program with respect to known parts of its input, the so-called *static* input. Running the specialized program on the remaining input (called the *dynamic* input) must yield the same result as running the original program on the complete input. Here we only describe the concepts of partial evaluation that are essential for this paper—a thorough account of partial evaluation can be found in the textbook of Jones et al. [14].

In this paper, we use offline partial evaluation. In offline partial evaluation, the process of partial evaluation is staged into (1) a *binding-time analysis* (BTA) and (2) the specialization of a program annotated with binding-time information. Binding-time analysis classifies as static the expressions that depend only on the static input; such expressions are evaluated during specialization. Expressions that also depend on the dynamic input are classified as dynamic, and are reconstructed to form the specialized program. The goal of a binding-time improvement is to rearrange the code so that more terms are classified as static.

In this paper, we use the Similix [5] partial evaluator for Scheme. Similix has the following notable features:

- *Monovariant BTA*: A monovariant BTA annotates each program construct with exactly one binding time.

- *Memoization of specialized code*: A memoizing specializer associates with each block of specialized code the set of static values with respect to which the code has been specialized. When the original code is to be specialized again with respect to the same static values, the specializer simply generates a function call or jump to the previously generated code. Similix considers a conditional expression with a dynamic test to be such a block of code, and these *memoization points* are identified during the

BTA. Other degrees of granularity are possible. Residual code originating from a memoization point is called a *variant* of this memoization point.

Even though the programs in this paper are presented in Scheme [18] (described briefly in Appendix 4.A), no Scheme-specific features are used; we expect the results to carry over directly to other functional languages and memoizing offline partial evaluators with a standard monovariant BTA.

4.3 Straightforward implementation of a string matcher

A pattern string `p` appears in a data string `d` if `p` is the prefix of some suffix of `d`. Thus, a straightforward algorithm to match a pattern `p` against a data string `d` is to proceed as follows: compare `p` against the prefix of `d`; if the match fails, restart, by trying to match against the tail of `d`. Figure 4.1 shows a direct implementation of this algorithm, where a string is represented as a list of symbols: procedure `main` takes a pattern `p` and a data string `d`; it calls the procedure `match`, passing `p` and `d` also to additional parameters `pp` and `dd`. The match of the prefix is performed by traversing `p` and `d`; `pp` and `dd` are used to restart the matching process on the original pattern and the tail of the current data string in the case of a mismatch.

The straightforward implementation has a running time of $O(|\texttt{p}| \cdot |\texttt{d}|)$. A worst-case example is matching `'(a a b)` against `'(a a a a a a b)`: The complete pattern is repeatedly matched against the data string. Figure 4.2 shows the result of specializing the straightforward algorithm with respect to `'(a a b)`:[1] Similix simply unfolds the static recursion on `p` and no noteworthy efficiency gain is achieved.

The source of the factor of $|\texttt{p}|$ in the complexity of the implementation of Figure 4.1 is the fact that this implementation loses information about the prefix of the data string with every restart: if k characters of the pattern have been successfully matched against the data string, and a restart occurs, then the first $k-1$ characters of the data string on which the matcher is restarted are already known. This kind of information is called *positive information*, because it originates from *successful* equality tests (the third conditional branch in Figure 4.1).

4.4 Pattern matching with positive information

Our derivation begins with the straightforward implementation of a matcher from Figure 4.1. We rewrite this implementation by exploiting information that can be deduced from the truth of dynamic conditional tests (*positive information*). Specialization of the resulting program with respect to a pattern of length

[1] In order to improve readability, in the output of Similix we have substituted more intuitive function names and changed local to global definitions.

```
(define (main p d)
  (match p d p d))

(define (match p d pp dd)
  (cond
    [(null? p) 'accept]                          ; matched the complete pattern
    [(null? d) 'reject]          ; reached end of text without complete match
    [(equal? (car p) (car d))   ; continue matching (cdr p) against (cdr d)
     (match (cdr p) (cdr d) pp dd)]
    [else                                ; restart, matching pp against (cdr dd)
     (match pp (cdr dd) pp (cdr dd))]))
```

Figure 4.1: A straightforward implementation of a string matcher

```
(define (main-0 d_0) (match_aab d_0 d_0))

(define (match_aab d_1 dd_0)
  (cond
    [(null? d_1) 'reject]
    [(equal? 'a (car d_1)) (match_ab (cdr d_1) dd_0)]
    [else (match_aab (cdr dd_0) (cdr dd_0))]))

(define (match_ab d_1 dd_0)
  (cond
    [(null? d_1) 'reject]
    [(equal? 'a (car d_1)) (match_b (cdr d_1) dd_0)]
    [else (match_aab (cdr dd_0) (cdr dd_0))]))

(define (match_b d_1 dd_0)
  (cond
    [(null? d_1) 'reject]
    [(equal? 'b (car d_1)) 'accept]
    [else (match_aab (cdr dd_0) (cdr dd_0))]))
```

Figure 4.2: The straightforward implementation, specialized to `'(a a b)`. A call `(match`$_x$ `d dd)` corresponds to `(match` x `d '(a a b) dd)`.

n produces a specialized program that consists of n comparisons and null-tests, and that performs at most $2m$ comparisons and at most $2m+1$ null-tests when applied to a string of length m.

4.4.1 Implementation

As pointed out in Section 4.3, the naive matcher loses information with every restart: if a restart occurs after k characters have been successfully matched,

then the first $k-1$ characters of the data string on which the matcher is restarted are already known (see Figure 4.3a). Instead of backtracking on the data string by "shifting" the pattern one position and matching it against the data string (Figure 4.3b), a matcher that collects the positive information can initially compare the pattern against the positive information (Figure 4.3c). While this modification does not improve the performance of the source program, it does improve the performance of the specialized program; because positive information is only dependent upon the pattern, partial evaluation can precompute the result of comparisons with this information.

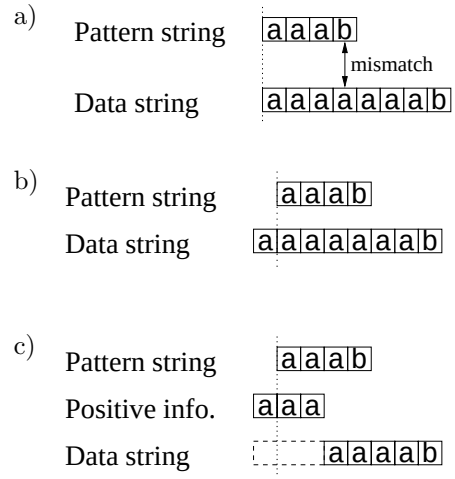

Figure 4.3: The role of positive information. a) A mismatch occurs. To the left of the mismatch, pattern string and data string are equal (positive information). b) Restart of the straightforward matcher. The pattern string is shifted one position to the right and matching resumes by comparing the pattern string with the data string. c) Restart of a matcher that specializes well. Positive information has been collected, and matching resumes by comparing the pattern string against it.

The straightforward matcher can be transformed into a matcher behaving as outlined above as follows:

1. We transform the implementation to make positive information explicit.

2. Static positive information is still lost because it is mixed with dynamic data. We remedy this by separating the affected static and dynamic values.

3. We reorder the algorithm such that decisions that depend only on static data are always made before examining dynamic data.

Step 1 is problem specific, while steps 2 and 3 amount to well-known binding-time improvements.

Making positive information explicit Figure 4.4 shows an implementation of the string matcher in which the arguments `pp` and `dd` have been replaced by the single argument `pi` ("positive information"). As shown in the third "cond" line, the argument `pi` records the characters that have been successfully matched. At all times, the original pattern is the value of `(append pi p)`, while the current position in the data is the value of `(append pi d)`. These invariants are used to rewrite the fourth "cond" line, which restarts the matching process on failure.

```
(define (main p d)
  (match p d '()))

(define (match p d pi)
  (cond
    [(null? p) 'accept]
    [(null? d) 'reject]
    [(equal? (car p) (car d))
     (match (cdr p) (cdr d) (append pi (list (car p))))]
    [else (match (append pi p) (cdr (append pi d)) '())]))
```

Figure 4.4: The string matcher with explicit positive information

While the identification and introduction of positive information is somewhat subtle, the use of such information appears to be critical to deriving efficient implementations of pattern matchers. The use of partial evaluation enables this information to be introduced while retaining a concise and monolithic implementation. While the second and third steps below somewhat compromise the conciseness of the implementation, they could, in principle, be carried out by an automatic (but user-guided) tool performing a collection of binding-time improvements.

Separating static and dynamic values The static positive information collected in `pi` does not survive a restart: `pi` is bound to `'()` and the concatenation of the positive information `pi` with the dynamic data string `d` produces a dynamic value. Our next step is a binding-time improvement to separate `(append pi d)`. This transformation is a variation of the technique of structure splitting that has been used to implement partially-static tuples [23]; here we modify this standard approach in order to represent a partially-static list. Essentially, we (1) replace `d` with an abstract data type that maintains the list as two components, with operations `null?`, `car`, `cdr` and `append`, and (2) inline the operations on this data type.

Concretely, we divide the dynamic parameter `d` into two parameters `s_d` ("static `d`") and `d_d` ("dynamic `d`"); the former corresponds to the `pi` portion of `(cdr (append pi d))` and the latter corresponds to the `d` portion of this value. The parameters `s_d` and `d_d` now represent the prefix and suffix of a single list, so we must rewrite each operation on the data string accordingly. We illustrate the transformation using the third "cond" line:

```
[(equal? (car p) (car d))
 (match (cdr p) (cdr d) (append pi (list (car p))))]
```

In this case, the `car` and `cdr` operations on `d` are each replaced by either an operation on `d_d` or on `s_d`, according to whether the prefix `s_d` is empty or not:

```
[(equal? (car p) (if (null? s_d) (car d_d) (car s_d)))
 (match (cdr p) (if (null? s_d) s_d (cdr s_d))        ; argument s_d
                (if (null? s_d) (cdr d_d) d_d)        ; argument d_d
                (append pi (list (car p))))]
```

Because `s_d` is static, the redundant `null?` tests are eliminated by partial evaluation.

Reordering control-flow decisions The previous transformation replaces a comparison between the head of the static pattern and the head of the dynamic data string with a comparison between the head of the static pattern and the result of a conditional expression. The conditional statically selects either the head of the static or the dynamic portion of the new representation of the data string. Because the result of this conditional expression is either static or dynamic, the binding-time analysis considers the result to be dynamic. Accordingly, the outer equality test becomes dynamic as well, even though it may be possible to determine its value based on the value of static subexpressions alone. To improve the binding times, we propagate the context of the conditional into its branches, as can be performed by continuation-passing-style specialization [4, 21]. The result is shown in Figure 4.5. Some static expressions involving `s_d` have been simplified by hand for readability; these simplifications could as well be carried out during specialization.

Compared with the matcher with explicit positive information (Figure 4.4), the code in Figure 4.5 looks rather complicated. Nevertheless, one should keep in mind that the transition from Figure 4.4 to Figure 4.5 is purely mechanical.

Specializing the string matcher from Figure 4.5 with respect to the pattern `'(a a b)` produces the residual program displayed in Figure 4.6. Now, when matching fails, the specialized program restarts the matching on either the current string or the tail of the current string, never backing up as done by the original implementation (*cf.* Figure 4.2).

Correctness Let $\texttt{main}_{\texttt{pos}}$ and $\texttt{match}_{\texttt{pos}}$ be the functions defined in the original implementation (Figure 4.1) and $\texttt{main}_{\texttt{orig}}$ and $\texttt{match}_{\texttt{orig}}$ be the functions defined in the derived implementation using positive information (Figure 4.5).

```
(define (main p d)
  (match p '() d '()))

(define (match p s_d d_d pi)
  (cond
    [(null? p) 'accept]
    [(null? s_d)                          ; no positive information available
     (cond
       [(null? d_d) 'reject]
       [(equal? (car p) (car d_d))
        (match (cdr p) '() (cdr d_d) (append pi (list (car p))))]
       [(null? pi) (match p '() (cdr d_d) '())]
       [else (match (append pi p) (cdr pi) d_d '())])]
    [else                                 ; positive information available
     (cond
       [(equal? (car p) (car s_d))
        (match (cdr p) (cdr s_d) d_d (append pi (list (car p))))]
       [else (match (append pi p) (cdr (append pi s_d)) d_d '())])]))
```

Figure 4.5: The string matcher, ready for specialization

```
(define (main-0 d_0) (match_|aab d_0))

(define (match_|aab d_d_0)
  (cond
    [(null? d_d_0) 'reject]
    [(equal? 'a (car d_d_0)) (match_a|ab (cdr d_d_0))]
    [else (match_|aab (cdr d_d_0))]))

(define (match_a|ab d_d_0)
  (cond
    [(null? d_d_0) 'reject]
    [(equal? 'a (car d_d_0)) (match_aa|b (cdr d_d_0))]
    [else (match_|aab d_d_0)]))

(define (match_aa|b d_d_0)
  (cond
    [(null? d_d_0) 'reject]
    [(equal? 'b (car d_d_0)) 'accept]
    [else (match_a|ab d_d_0)]))
```

Figure 4.6: The string matcher, specialized to `'(a a b)`. A call (`match`$_{x|y}$ `d_d`) corresponds to (`match` y `'() d_d` x).

We can then show by well-founded induction based on the lexicographic ordering $\langle |\texttt{d\_d}|, |\texttt{(append pi s\_d)}|, |\texttt{p}| \rangle$ that for all `p`, `s_d`, `d_d` and `pi`, evaluating

(match$_{\text{orig}}$ p d (append pi p) (append pi d)) (where d stands for the concatenation of s_d and d_d) and evaluating (match$_{\text{pos}}$ p s_d d_d pi) yields the same result (a formal proof is given in Appendix 4.B). It then follows directly that main$_{\text{pos}}$ and main$_{\text{orig}}$ are equivalent.

4.4.2 Complexity of the specialized code

We analyze the size and the running time of the specialized program. In both cases, it is helpful to distinguish the code that is residualized from the code that is evaluated during specialization. We thus refer to the result of the binding-time analysis by Similix, illustrated in Figure 4.7. The parameters of each function are annotated with their binding times. Basic function calls that are reconstructed during specialization are underlined; user-defined functions are always unfolded. Static expressions whose results have to be residualized are enclosed by underlined parenthesis. A comment marks a memoization point inserted by Similix.

```
(define (main p^s d^d)
  (match p '() d '()))

(define (match p^s s_d^s d_d^d pi^s)
  (cond
    [(null? p) 'accept]
    [(null? s_d)
     (cond                                          ; memoization point
       [(null? d_d) 'reject]
       [(equal? (car p) (car d_d))
        (match (cdr p) '() (cdr d_d) (append pi (list (car p))))]
       [else
        (cond
          [(null? pi) (match p '() (cdr d_d) '())]
          [else (match (append pi p) (cdr pi) d_d '())])])]
    [else
     (cond
       [(equal? (car p) (car s_d))
        (match (cdr p) (cdr s_d) d_d (append pi (list (car p))))]
       [else (match (append pi p) (cdr (append pi s_d)) d_d '())])]))
```

Figure 4.7: The annotated string matcher

It is easy to see that the size of the residual code is governed by the number of residualized conditionals; we therefore make the number of residualized null tests and comparisons (which directly corresponds to the number of residualized conditionals) our measure for the size of the residualized code. We measure complexity in terms of the length of input, assuming a finite alphabet of a given fixed size.

Size

The following theorem relates the size of the specialized code to the size of the pattern:

Theorem 4.1 *Specializing the implementation in Figure 4.5 with respect to a pattern of length n yields a residual program of size linear in n. More precisely, exactly n comparisons and null tests on the dynamic data are generated.*

For the proof of Theorem 4.1 we need the following lemma, which is proved straightforwardly by induction on the number of calls to `match`:

Lemma 4.2 *In the evaluation of* `(main p0 d0)` *(as defined in Figure 4.5), for any $k \geq 1$, in the k^{th} call to* `match`*, the concatenation of argument* `pi` *with argument* `p` *is equal to* `p0`*.*

We now turn to the proof of Theorem 4.1.

Proof: In `match`, there is only one occurrence of a `null?` test and one occurrence of an `equal?` test that are reconstructed during specialization (as shown by the underlined constructs in Figure 4.7). Both are in the scope of the memoization point in `match`. Thus, the number of residual tests depends on the number of variants generated.

Analyzing the guards of the outer cond-expression in the definition of `match` in Figure 4.7 shows that the memoization point is reached only if `s_d` = `'()`. Hence the number of possible configurations of the static data at the memoization point is governed by the contents of `p` and `pi`. Lemma 4.2 implies that there are at most $n + 1$ such configurations. All of them are indeed reached during partial evaluation; however only n of them will be encountered at the memoization point, because with `p` = `'()`, it cannot be reached. □

Execution time

We measure the execution time in terms of the number of equality and null tests performed by the algorithm:

Theorem 4.3 *Let* `main_res` *be the code generated by specializing* `(main p d)` *with respect to* `p`*. Let t_e be the number of equality tests and t_n be the number of null tests performed when applying* `main_res` *to any* `d`*. Then $t_e \leq 2|\texttt{d}|$ and $t_n \leq t_e + 1$.*

Theorem 4.3 follows directly from the following lemma:

Lemma 4.4 *Let* `match_res` *be the code generated by specializing* `(match p s_d d_d pi)` *with respect to* `p`*,* `s_d` *and* `pi`*. Let t_e be the number of equality tests and t_n the number of null tests performed by applying* `match_res` *to any* `d_d`*. Then, $t_e \leq 2|\texttt{d\_d}| + |\texttt{s\_d}| + |\texttt{pi}|$ and $t_n \leq t_e + 1$.*

Proof: The equality and null tests performed by `match_res` are the same as the equality and null tests that are performed by `(match p s_d d_d pi)` and that are marked as irreducible (underlined in Figure 4.7) by the binding-time analysis. Hence we can establish the bounds t_e and t_n by analyzing the number of such tests performed during the evaluation of `(match p s_d d_d pi)`. The proof is by well-founded induction over the evaluation of `(match p s_d d_d pi)` based on the lexicographic ordering $\langle|\texttt{d\_d}|, |\texttt{(append pi s\_d)}|, |\texttt{p}|\rangle$. □

4.5 Pattern matching with both positive and negative information

Even though the result of partially evaluating the string matcher from Figure 4.5 has a pattern-independent running time linear in the size of the data string, its behavior is not optimal; it may repeatedly compare a character of the data string with the same character. This behavior is illustrated by the specialized code in Figure 4.6. In $\texttt{match}_{\texttt{a|ab}}$, if the first character of the data string is not 'a', $\texttt{match}_{\texttt{|aab}}$ is called, which again compares the first character of the data string to 'a'.

Redundant comparisons can be avoided by using *negative* information, i.e., information about which comparisons have already failed for the current first character of `d_d`. We use techniques similar to those presented in Section 4.4.1, now to exploit negative information such that the specialized code does not perform redundant comparisons.

4.5.1 Implementation

We transform the algorithm from Figure 4.5 in three steps. First, we add negative information. Then, we use the negative information to avoid dynamic operations when their result can be deduced statically from the negative information. As before, this change only improves the result of specialization after we reorganize conditionals to improve the binding times.

Making negative information explicit We introduce an additional argument `ni` that contains negative information. Whenever a mismatch between a character c from the pattern and the head of the dynamic data `d_d` occurs, c is added to `ni`. Whenever the first character of `d_d` is thrown away, the information collected in `ni` is outdated and therefore `ni` is set to `'()`. Figure 4.8 shows the modified parts of the implementation—the changes have been underlined.

Using negative information The following invariant can be shown by induction on the number of calls to `match`:

```
(define (main p d)
  (match p '() ni d '()))

(define (match p s_d ni d_d pi)
  (cond
    [(null? p) ...]
    [(null? s_d)                              ; no positive information available
     (cond
       [(null? d_d) 'reject]
       [(equal? (car p) (car d_d))
        (match (cdr p) '() '() (cdr d_d) (append pi (list (car p))))]
       [(null? pi) (match p '() '() (cdr d_d) '())]
       [else
        (match (append pi p) (cdr pi) (cons (car p) ni) d_d '())])]
    [else ...]))                              ; positive information available
```

Figure 4.8: Excerpts of a string matcher that maintains negative information (underlined code computes negative information)

Lemma 4.5 *In the evaluation of* `(main p0 d0)` *(as defined in Figure 4.8), for any* $k \geq 1$*, in the* k^{th} *call to* `match`*,* `ni` $\neq$ `'()` *implies that* `d_d` $\neq$ `'()` *and* `(car d_d)` $\notin$ `ni`*.*

Using this invariant, we transform the code of Figure 4.8 to only perform null tests on `d_d` and comparisons with `(car d_d)` when their outcome cannot be determined from the negative information. Specifically, the test `(null? d_d)` becomes `(and (null? ni) (null? d_d))` (if there is negative information, `d_d` is guaranteed to be nonempty), and the test

```
(equal? (car p) (car d_d))
```

becomes

```
(and (not (member (car p) ni)) (equal? (car p) (car d_d)))
```

(any character contained in the negative information cannot appear as the head of `d_d`).

Reordering control-flow decisions The previous transformation creates "and" expressions where the first argument is static and the second argument is dynamic. By the semantics of "and", it may be possible to determine the value of such an expression based on the value of the first (static) expression alone. Thus, we improve the binding times by separating these tests; the result is shown in Figure 4.9.

Correctness Correctness can be shown as for the implementation using only positive information. Here, the relationship between the original and optimized versions of `match` only holds under the condition that `ni` $\neq$ `'()` implies

```
(define (main p d)
  (match p '() '() d '()))

(define (match p s_d ni d_d pi)
  (cond
    [(null? p) 'accept]
    [(and (null? s_d) (null? ni))          ; no static information available
     (cond
       [(null? d_d) 'reject]
       [(equal? (car p) (car d_d))
        (match (cdr p) '() '() (cdr d_d) (append pi (list (car p))))]
       [(null? pi) (match p '() '() (cdr d_d) '())]
       [else (match (append pi p) (cdr pi) (list (car p)) d_d '())])]
    [(null? s_d)                           ; negative information available
     (cond
       [(member (car p) ni)
        (if (null? pi)
            (match p '() '() (cdr d_d) '())
            (match (append pi p) (cdr pi) ni d_d '()))]
       [(equal? (car p) (car d_d))
        (match (cdr p) '() '() (cdr d_d) (append pi (list (car p))))]
       [(null? pi) (match p '() '() (cdr d_d) '())]
       [else
        (match (append pi p) (cdr pi) (cons (car p) ni) d_d '())])]
    [else                                  ; positive information available
     (cond
       [(equal? (car p) (car s_d))
        (match (cdr p) (cdr s_d) ni d_d (append pi (list (car p))))]
       [else
        (match (append pi p) (cdr (append pi s_d)) ni d_d '())])]))
```

Figure 4.9: A string matcher that uses both positive and negative information, ready for specialization

`d_d` $\neq$ `'()` and `(car d_d)` $\notin$ `ni` (a formal proof is given in Appendix 4.C). By straightforward induction over the number of recursive calls, we can further show that the optimized matcher performs no redundant equality tests: the argument `ni` always contains all the characters that `(car d_d)` has been matched against in the previous calls, and matches against `(car d_d)` occur only for characters not in `ni`.

4.5.2 Complexity of the specialized code

Again, we analyze the size of the specialized program and its running time. Using the same techniques as in Section 4.4.2, we show the following results about the size and execution time of the specialized code:

Theorem 4.6 (Size) *Specializing the algorithm from Figure 4.9 with respect to a pattern of length n containing c distinct characters yields a residual program with at most n null tests and $n \cdot c$ equality tests.*

Theorem 4.7 (Execution time) *Let* $\texttt{main}_{\texttt{neg}}\texttt{\_res}$ *be the result of specializing* $(\texttt{main}_{\texttt{neg}}\ \texttt{p}\ \texttt{d})$ *with respect to* `p`*, and* $\texttt{main}_{\texttt{pos}}\texttt{\_res}$ *be the result of specializing* $(\texttt{main}_{\texttt{pos}}\ \texttt{p}\ \texttt{d})$ *with respect to* `p`*. When applying* $\texttt{main}_{\texttt{neg}}\texttt{\_res}$ *to* `d`*, the following holds:*

1. *At most as many equality and null tests are performed as when applying* $\texttt{main}_{\texttt{pos}}\texttt{\_res}$ *to* `d`*.*
2. *At most* $|\texttt{d}| + 1$ *null tests are performed.*

The actual proofs are deferred to Appendix 4.D.

4.6 Variants

So far, we have derived two implementations of the naive KMP algorithm that specialize well using standard partial-evaluation techniques. Previously, other implementations have been proposed that have similar properties [2, 7, 14, 28]. In this section, we explore the relationship between our implementation and the variants proposed by Consel and Danvy [7] and by Jones et al. [14], as well as other possible variants.

4.6.1 Linguistic variants

Our implementation can be characterized as using a single loop expressed as a recursive equation, in which all of the arguments are disjoint, all of the recursive calls are in tail position, the positive information is accumulated and maintained in an auxiliary list, and (optionally) a set of negative information is maintained. This characterization suggests that alternative implementations can be derived by varying the following characteristics:

- Disjoint parameters *vs.* overlapping parameters (this point is rather technical, but is explained in Section 4.6.2 below).
- A single loop processing both the dynamic data string and the static positive information *vs.* one loop processing the dynamic data string and another processing the static positive information.
- Tail recursion *vs.* non-tail recursive calls.
- Accumulating positive information *vs.* reconstructing it where needed.
- Maintaining the positive information using a list *vs.* maintaining the positive information as an offset into the pattern.
- Recording 0, 1, or a set of characters of negative information.
- Recursive equations *vs.* block structure.

4.6.2 Overlapping parameters

We first consider the implementation of Jones et al. While this implementation postdates the implementation of Consel and Danvy, it can be derived from our implementation more simply, by adding extra, overlapping parameters.

In our implementation, the parameters `pi` and `p` maintain disjoint partitions of the pattern, while `pi` and `s_d` maintain disjoint partitions of the positive information. This approach implies that to access the complete pattern, we must append `pi` and `p`, and to access the complete positive information, we must append `pi` and `s_d`. An alternative is to maintain the values `(append pi p)` and `(append pi s_d)` as extra parameters `pat` (for *pattern*) and `pos` (for *positive information*). The values of these parameters *overlap*, because both contain the value of `pi`. Their presence makes `pi` redundant: every remaining occurrence either is used to rebind the parameter `pi` or can be replaced by `pos`, because `s_d` is known to be empty. Figure 4.10 shows the result of adding parameters `pat` and `pos`, and removing parameter `pi`.

```
(define (main p d)
  (match p '() d p '()))

(define (match p s_d d_d pat pos)
  (cond
    [(null? p) 'accept]
    [(null? s_d)                          ; no positive information available
     (cond
       [(null? d_d) 'reject]
       [(equal? (car p) (car d_d))
        (match (cdr p) '() (cdr d_d) pat (append pos (list (car p))))]
       [(null? pos) (match p '() (cdr d_d) pat '())]
       [else (match pat (cdr pos) d_d pat (cdr pos))])]
    [else                                 ; positive information available
      (cond
        [(equal? (car p) (car s_d))
         (match (cdr p) (cdr s_d) d_d pat pos)]
        [else (match pat (cdr pos) d_d pat (cdr pos))])]))
```

Figure 4.10: Arguments `pat` and `pos` are added, argument `pi` is eliminated

Because this implementation maintains static information in a form isomorphic to the use of static information in our implementation, the specialized code is unchanged. Jones et al. also present a version using negative information, which can be derived using the same techniques as presented in Section 4.5.1.

4.6.3 Towards Consel and Danvy's implementation

The implementation proposed by Consel and Danvy can be derived from ours by adding most of the identified linguistic variations, i.e, overlapping parame-

ters, two loops, non-tail recursion, reconstructing indices and one character of negative information. We describe each step in one possible derivation path, beginning with the implementation of Jones et al., which already uses overlapping parameters.

Two loops and non-tail recursion

In the variants we have explored so far, there is a single loop, matching the pattern to either the static positive information or the dynamic data string. Nevertheless, the computation performed and the binding-time properties of these two cases are quite distinct, and the emptiness of `s_d` is essentially used as a flag to distinguish between the two. Thus, it can be illuminating to manually separate the implementation into specific functions `match_d` and `match_s` that address each case.

Because Similix inserts memoization points only at dynamic conditionals, this transformation is merely cosmetic if Similix is used. Another strategy is to add a memoization point at the top of any function that contains a dynamic conditional. Following this strategy, in the monolithic implementation, every call to `match` would be a memoization point, introducing a chain of trivial function calls. Splitting the implementation, producing the completely static `match_s` function, avoids this problem.

Intuitively, `match_s` corresponds to the failure table used in the standard KMP algorithm. This intuition can be made more obvious by slightly modifying the code such that `match_d` calls `match_s` non–tail-recursively and `match_s` returns a result instead of calling `match_d`. The result of carrying out these transformations on the variant with overlapping parameters is shown in Figure 4.11.

Reconstructing positive information and using indices

All variants so far accumulate positive information by appending characters to additional arguments such as `pos`. It is easy to show that `(append pos p)` is equal to `pat` for every call to `match_d` in Figure 4.11; therefore, whenever `pos` is needed, it can be reconstructed by collecting the first $|\texttt{pat}| - |\texttt{p}|$ characters of `pat`. Further, instead of explicitly constructing the positive information by copying a prefix of the pattern `pat`, one can use `pat` as it is, using an additional parameter to keep track of the length of the prefix that corresponds to the positive information.

Consel and Danvy's implementation

Consel and Danvy [7] present an implementation of a string matcher that (1) avoids redundant null tests on `d_d`, and (2) uses negative information in a special case.

Avoiding redundant null-tests In Figure 4.11, recursive calls of the dynamic loop `match_d` to itself with an unchanged parameter `d_d` lead to

```
(define (main p d)
  (match_d p d p '()))

(define (match_d p d_d pat pos)        ; no positive information is available
  (cond
    [(null? p) 'accept]
    [(null? d_d) 'reject]
    [(equal? (car p) (car d_d))
     (match_d (cdr p) (cdr d_d) pat (append pos (list (car p))))]
    [(null? pos) (match_d p (cdr d_d) pat '())]
    [else
      (let ([p-pos (match_s pat (cdr pos) pat (cdr pos))])
        (match_d (car p-pos) d_d pat (cadr p-pos)))]))

(define (match_s p s_d pat pos)    ; positive information might be available
  (cond
    [(null? s_d) (list p pos)]
    [(equal? (car p) (car s_d)) (match_s (cdr p) (cdr s_d) pat pos)]
    [else (match_s pat (cdr pos) pat (cdr pos))]))
```

Figure 4.11: `match` is split and non-tail recursion is added

a redundant null-test: we know `d_d` to be nonempty. By moving null-tests on `d_d` and `p` to additional entry-points for the dynamic loop and choosing the appropriate entry-point according to what is known about `d_d` and `p`, we avoid such tests.

A special case of using negative information In Figure 4.11, when calling `match_s` from `match_d`, it has already been determined that `(car p)` is different from `(car d_d)`. Thus, if the first character of the pattern part returned by `match_s` is equal to `(car p)`, the comparison with `(car d_d)` performed by the recursive call to `match_d` is guaranteed to fail. Instead of calling `match_d`, we therefore can attempt to restart the matching process on `(cdr d_d)`. However, there is no convenient way to restart the matching process except in the case where the complete pattern needs to be matched against `(cdr d_d)`. For this special case, restarting the matching on `(cdr d_d)` is straightforward.

Figure 4.12 displays the result of modifying the code from Figure 4.11 as described above (also, positive information is reconstructed and indices are used, as described in Section 4.6.3). The code is very similar to Consel and Danvy's implementation. They, however, additionally split `match_d` into two functions, distinguishing whether positive information has been accumulated (i.e., $\texttt{p} \neq \texttt{pat}$) or not. The transformation does not affect the result of specialization and we therefore omit it.

```
(define (main p d)                              ;  p and  d might be empty
  (cond [(null? p) 'accept]
        [else (match_d_orig_p p d)]))

(define (match_d_orig_p p d)                      ; only  d might be empty
  (cond [(null? d) 'reject]
        [else (match_d p d p)]))

(define (match_d p d_d pat)             ; neither  p and  d might be empty
  (cond
    [(equal? (car p) (car d_d))
     (cond [(null? (cdr p)) 'accept]
           [(null? (cdr d_d)) 'reject]
           [else (match_d (cdr p) (cdr d_d) pat)])]
    [(equal? p pat) (match_d_orig_p p (cdr d_d))]
    [else
     (let* ([pos_len (- (length pat) (length p))]
            [np (match_s pat (cdr pat) (- pos_len 1)
                         pat (cdr pat) (- pos_len 1))])
       (if (and (equal? np pat) (equal? (car np) (car p)))
           (match_d_orig_p p (cdr d_d))
           (match_d np d_d pat)))]))

(define (match_s p s_d s_d_len pat pos pos_len)
  (cond
    [(= s_d_len 0) p]
    [(equal? (car p) (car s_d))
     (match_s (cdr p) (cdr s_d) (- s_d_len 1) pat pos pos_len)]
    [else
     (match_s pat (cdr pos) (- pos_len 1)
              pat (cdr pos) (- pos_len 1))]))
```

Figure 4.12: Redundant null tests are avoided and negative information is used in a special case

Consel and Danvy also present a matcher that maintains exactly one character of negative information. The following lemma, which is easily proven by well-founded induction based on $\langle |\texttt{d\_d}|, |\texttt{(append pi s\_d)}|, |\texttt{p}| \rangle$, shows how one character of negative information already is maintained "automatically" in `match_s`:

Lemma 4.8 *In the evaluation of* `(main p0 d0)` *(as defined in Figure 4.12), whenever* `match_s` *is called, the* $(\texttt{s\_d\_len} + 1)^{\text{st}}$ *character of* `s_d` *is different from* `(car d_d)`.

This information can be used when `match_s` is about to return `p` such that `(car p)` is equal to the $(\texttt{s\_d\_len} + 1)^{st}$ character of `s_d`: Instead of returning

`p`, `match_s` continues matching against the remaining positive information.

4.7 Related work

Consel and Danvy [7] conceived a binding-time improvement that makes it possible to generate an efficient matcher from a naive matcher by using standard partial-evaluation techniques. Their insight was to exploit the relationship between the pattern string and the dynamic data, as exemplified by the diagram of Figure 4.3 (a similar diagram appeared in their paper). Based on this insight, further investigations into the derivation of efficient matchers using partial evaluation have been carried out. Jones et al. [14] and Amtoft [2] present a modified version of a matcher that standard partial evaluation can specialize into an efficient matcher. Sørensen et al. [28] present a version that is basically equivalent to the former of these variants. Amtoft et al. [3] parameterize a string matcher over a static cache and present instantiations for which partial evaluation produces KMP-like and Boyer-Moore–like string matchers. Gomard [12] and Kaneko and Takeichi [17] show how to generate efficient matchers with variants of a specialization scheme called *fully lazy evaluation*. Jørgensen [16] and Sestoft [26] apply the combination of partial evaluation and explicit encoding of positive and negative information to implement efficient matching of a data structure against a collection of patterns, as is used in implementing the `case` construct of languages such as ML and Haskell. What amounts to positive information has also been used in hand-coded pattern-matching compilers [24, Chapter 5].

Other work on applying partial evaluation to string matching is mainly concerned with identifying the additional features that must be added to the standard partial-evaluation framework in order to pass the KMP-test: Instead of making positive/negative information explicit in the source program, one can use a specializer with the capability of collecting and using such information. For example Sørensen et al. [28] observe that positive supercompilation [10, 11] maintains more information during the transformation process than does partial evaluation. Nevertheless only positive information is maintained; positive supercompilation of a naive string matcher and a pattern results in a linear matcher that may perform redundant tests. In contrast, Smith [27] observes that a partial evaluator for a family of constraint logic programming languages succeeds in generating linear matchers that do not perform redundant tests, because negative information is maintained as well. The same is true for Generalized Partial Computation [8], where a theorem prover is used to derive additional information from the truth or falsity of enclosing conditional tests, and for Queinnec and Geffroy's intelligent backtracking system [25], where abstract descriptions of the matched and unmatched patterns are maintained across success and failure continuations. Other partial evaluators that pass the KMP-test include partial deduction [22] and partial evaluators for functional logic programs [1, 20]. A generic way to make a given partial evaluator more powerful is the *interpretive approach* where an interpreter is inserted between a partial

evaluator and a source program. Glück and Jørgensen [9] show that composing an interpreter that propagates positive information with Similix produces a positive supercompiler that passes the KMP test.

Whereas systems capable of collecting and using positive and negative information successfully pass the KMP test, they are likely to be defeated by related but more complicated problems. For example, Amtoft et al. [3] observe that to derive Boyer-Moore–like matchers, information also needs to be thrown away; different strategies for discarding information yield different matchers. Handling such problems therefore requires modification of the input program not only for standard partial evaluators, but also for most, if not all, of the more powerful systems mentioned above.

4.8 Conclusion

Specializing string matchers is a canonical example of partial evaluation. Nevertheless, it has been found that in string matchers that specialize well under *standard* partial evaluation, statically-determinable implicit information about the dynamic input has to be represented explicitly. Numerous implementations of such string matchers have been developed, usually by starting with a naive string matcher and applying binding-time improvements to it. The published presentations, however, generally carry out the binding-time improvements atomically, and thus do not show how such binding-time improvements can be achieved in a systematic way. Further, as Amtoft put it, "it is not obvious that [the transformation applied to the naive string matcher] preserves semantics" [2, p. 176].

We have presented a stepwise derivation of a string matcher that makes positive information explicit in the static data, and that thus specializes well using standard partial-evaluation techniques. We have also derived an extension that makes negative information explicit; as a consequence, redundant tests in the specialized code are eliminated. In both cases, we have proved that the size of the specialized program is linear in the size of the pattern and the running time of the specialized program is linear in the size of the data string. We have further explored the relationship between our implementation and alternative implementations: we identified properties of our implementation that can be varied without changing the overall effect of partial evaluation and derived several variants of our implementation. In particular, we have shown how to derive two of the published implementations [7, 14], thus providing a conceptual link between them and a wide range of variants, all of which specialize well under standard partial evaluation.

Acknowledgements

We thank Olivier Danvy for suggesting we explore the implementation variants analyzed in Section 4.6, and for helpful comments and encouragement

throughout this work. We also thank Charles Consel and Riko Jacob for useful comments on this paper.

4.A An overview of Scheme

Scheme is a call-by-value, dynamically-typed, statically-scoped dialect of Lisp. In this appendix, we give a brief overview of the features of Scheme, restricted to their use in this paper.

A Scheme term is either a symbol, a number, or a list. A symbol is a sequence of characters, such as `match`. A list is denoted by an open parenthesis followed by a sequence of terms, separated by whitespace, followed by a close parenthesis. Square brackets `[` and `]` may be used in place of parentheses. Programs are represented as terms, such as `x` or `(cons 3 x)`. A symbol represents a variable reference. A list indicates an application, in which the first element is the applied operator, and the remaining elements are the arguments. Data is constructed by quoting a term, as in `'accept` (the symbol `accept`), `'()` (the empty list), and `'(cons 5 x)` (a list consisting of the symbol `cons`, the number `5`, and the symbol `x`). Note the difference between `(cons 5 x)` and `'(cons 5 x)`: the former is interpreted by Scheme as a program, the latter constitutes a piece of data. In this paper, we use quoted lists of symbols to represent the pattern and data strings.

The application of most operators in Scheme is performed following a call-by-value strategy, and the application of most operators returns a value. We refer to operators that satisfy these two properties as *functions*. Some operators return no value. Some built-in operators, such as the conditional operator `if`, the local-binding operator `let`, and the boolean operator `and`, do not necessarily evaluate all of their arguments. We refer to operators that do not necessarily evaluate all of their arguments as *special forms*.

Global functions are defined using the special form `define`:

`(define (`*fn-name* arg_1 arg_2 `...)` *body*`)`

As shown by the example, `define` has two arguments: a list, of which the first element is the function name and the remaining elements are the parameter names, and the body of the function definition.

The list-processing functions we use are as follows (shown with arguments):

- `(null?` l`)`: returns true if the value of l is the empty list, and false otherwise.
- `(cons` a l`)`: constructs a list with the value of a as the first element, and the elements of l as the remaining elements.
- `(car` l`)`: returns the first element of the list l.
- `(cdr` l`)`: returns the a list containing all of the elements of l, except the first one.

- `(append` l_1 l_2`)`: constructs a list that contains all of the elements of l_1, followed by all of the elements of l_1.
- `(member` a l`)`: returns true if the value of a is an element of the list l, and false otherwise.
- `(list` e_1 e_2 `...)`: constructs a list consisting of the elements e_1, e_2, etc. The function `list` can take any number of arguments, including zero.
- `(length` l`)`: returns the number of elements in the list l.

We also use the function `equal?`, which tests any two values for equality, the function `=`, which tests two numerical values for equality, the function `-`, which performs subtraction, and the special form `and`, which returns true if both of its arguments are true, and false otherwise.

Conditionals are specified using an `if` expression, of the following form:

```
(if test consequent alternate)
```

If the value of the expression *test* is true, then the term *consequent* is evaluated; otherwise the term *alternate* is evaluated. A sequence of conditionals can be abbreviated using a `cond` term, having the following form:

```
(cond [test_1 exp_1]
      [test_2 exp_2]
      ...
      [else exp_n])
```

The terms $test_1$, $test_2$, *etc.* are evaluated in order until one is true, in which case the corresponding expression is evaluated. If none of the test expressions evaluates to true, then the expression corresponding to the `else` expression is evaluated. An `else` line is not needed when the sequence of tests is exhaustive. For readability, we use square brackets rather than parentheses to delimit the test-expression pairs.

Local variables are introduced using a `let` expression, having the following form:

```
(let ([var_1 exp_1]
      ...
      [var_n exp_n])
   exp)
```

The first argument to `let` is a list of pairs associating variables to expressions. These variables are bound to the values of the corresponding expressions during the evaluation of the body *exp*. The bindings to the variables $var_1, \ldots, var_n$ are not visible during the evaluation of any of the $exp_1, \ldots, exp_n$. The special form `let*` is a variant of `let`, in which the bindings preceeding `[`var_i exp_i`]` are visible in the evaluation of exp_i. For readability, we use square brackets rather than parentheses to delimit the variable-expression pairs.

4.B Correctness of the derived implementation using positive information

Let $\texttt{match}_{\texttt{orig}}$ be the original implementation (Figure 4.1 on page 103) and $\texttt{match}_{\texttt{pos}}$ be the derived implementation using positive information (Figure 4.5 on page 107). We show that for all `p`, `s_d`, `d_d` and `pi`, evaluating

```
(match_orig p d (append pi p) (append pi d))
```

(where `d` stands for the concatenation of `s_d` and `d_d`), and evaluating

```
(match_pos p s_d d_d pi)
```

yields the same result.

Proof: The proof is by induction on the tuple $\langle|\texttt{d\_d}|, |\texttt{(append pi s\_d)}|, |\texttt{p}|\rangle$, ordered lexicographically. It is straightforward to show that this value decreases at every function call of match_{pos}. We thus do not explicitly check that the induction hypothesis applies in each case.

For conciseness, we rewrite the Scheme code using a more mathematical notation. In particular, @ replaces `append`, [] replaces `'()`, and `hd` and `tl` replace `car` and `cdr`, respectively. We also implicitly rely on the associativity of `append`.

We want to show that for all p, s_d, d_d and pi,

$$\text{match}_{pos}(p, s_d, d_d, pi) = \text{match}_{orig}(p, s_d@d_d, pi@p, pi@s_d@d_d).$$

In all cases where $p = []$, the calls to match_{pos} and match_{orig} evaluate to *accept*. Otherwise, assume that the theorem is true for all smaller tuples. We proceed with an exhaustive case distinction. In the following, we assume for every case that none of the preceeding cases holds.

Consider the case $s_d = []$:

- $d_d = []$: In this case, the call to match_{pos} evaluates to *reject*. If both s_d and d_d are empty, then the second argument of match_{orig} is empty as well, and the call to match_{orig} also evaluates to *reject*.
- $\mathsf{hd}(p) = \mathsf{hd}(d_d)$: In this case the call to match_{pos} evaluates to

  $$\text{match}_{pos}(\mathsf{tl}(p), [], \mathsf{tl}(d_d), pi@[\mathsf{hd}(p)]),$$

  and the call to match_{orig} evaluates to

  $$\text{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(d_d), pi@p, pi@d_d).$$

  Both are equal:

  $$\begin{array}{ll} & \text{match}_{pos}(\mathsf{tl}(p), [], \mathsf{tl}(d_d), pi@[\mathsf{hd}(p)]) \\ \overset{\text{IH}}{=} & \text{match}_{orig}(\mathsf{tl}(p), []@\mathsf{tl}(d_d), pi@[\mathsf{hd}(p)]@\mathsf{tl}(p), pi@[\mathsf{hd}(p)]@[]@\mathsf{tl}(d_d)) \\ = & \text{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(d_d), pi@p, pi@d_d) \end{array}$$

- $pi = [\,]$: In this case, the call to match_{pos} evaluates to

  $$\text{match}_{pos}(p, [\,], \mathsf{tl}(d_d), [\,]),$$

  and the call to match_{orig} evaluates to

  $$\text{match}_{orig}(p, \mathsf{tl}(d_d), p, \mathsf{tl}(d_d)).$$

  Both are equal:

  $$\begin{array}{ll} & \text{match}_{pos}(p, [\,], \mathsf{tl}(d_d), [\,]) \\ \overset{\text{IH}}{=} & \text{match}_{orig}(p, [\,]@\mathsf{tl}(d_d), [\,]@p, [\,]@[\,]@\mathsf{tl}(d_d)) \\ = & \text{match}_{orig}(p, \mathsf{tl}(d_d), p, \mathsf{tl}(d_d)) \end{array}$$

- $pi \neq [\,]$: In this case, the call to match_{pos} evaluates to

  $$\text{match}_{pos}(pi@p, \mathsf{tl}(pi), d_d, [\,]),$$

  and the call to match_{orig} evaluates to

  $$\text{match}_{orig}(pi@p, \mathsf{tl}(pi@d_d), pi@p, \mathsf{tl}(pi@d_d)).$$

  Both are equal:

  $$\begin{array}{ll} & \text{match}_{pos}(pi@p, \mathsf{tl}(pi), d_d, [\,]) \\ \overset{\text{IH}}{=} & \text{match}_{orig}(pi@p, \mathsf{tl}(pi)@d_d, [\,]@pi@p, [\,]@\mathsf{tl}(pi)@d_d) \\ = & \text{match}_{orig}(pi@p, \mathsf{tl}(pi@d_d), pi@p, \mathsf{tl}(pi@d_d)) \end{array}$$

Now we turn to the case $s_d \neq [\,]$:

- $\mathsf{hd}(p) = \mathsf{hd}(s_d)$: In this case, the call to match_{pos} evaluates to

  $$\text{match}_{pos}(\mathsf{tl}(p), \mathsf{tl}(s_d), d_d, pi@[\mathsf{hd}(p)]).$$

  Because the first argument of match_{orig} is p and the second argument of match_{orig} is $s_d@d_d$, the constraint $\mathsf{hd}(p) = \mathsf{hd}(s_d)$ implies that the third cond line of match_{orig} is satisfied and the result is the value of

  $$\text{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(s_d@d_d), pi@p, pi@s_d@d_d).$$

  Both are equal:

  $$\begin{array}{ll} & \text{match}_{pos}(\mathsf{tl}(p), \mathsf{tl}(s_d), d_d, pi@[\mathsf{hd}(p)]) \\ \overset{\text{IH}}{=} & \text{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(s_d)@d_d, pi@[\mathsf{hd}(p)]@\mathsf{tl}(p), pi@[\mathsf{hd}(p)]@\mathsf{tl}(s_d)@d_d) \\ = & \text{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(s_d@d_d), pi@p, pi@s_d@d_d) \end{array}$$

- $\mathsf{hd}(p) \neq \mathsf{hd}(s_d)$: In this case the call to match_{pos} evaluates to

$$\text{match}_{pos}(pi@p, \mathsf{tl}(pi@s_d), d_d, [\,]),$$

and the call to match_{orig} evaluates to

$$\text{match}_{orig}(pi@p, \mathsf{tl}(pi@s_d@d_d), pi@p, \mathsf{tl}(pi@s_d@d_d)).$$

Both are equal:

$$\begin{array}{ll} & \text{match}_{pos}(pi@p, \mathsf{tl}(pi@s_d), d_d, [\,]) \\ \overset{\text{IH}}{=} & \text{match}_{orig}(pi@p, \mathsf{tl}(pi@s_d)@d_d, [\,]@pi@p, [\,]@\mathsf{tl}(pi@s_d)@d_d) \\ = & \text{match}_{orig}(pi@p, \mathsf{tl}(pi@s_d@d_d), pi@p, \mathsf{tl}(pi@s_d@d_d)) \end{array}$$

□

4.C Correctness of the derived implementation using negative information

Let $\texttt{match}_{\texttt{orig}}$ be the original implementation (Figure 4.1 on page 103) and let $\texttt{match}_{\texttt{neg}}$ be the derived implementation of `match` using positive and negative information (Figure 4.9 on page 112). We show that for all `p`, `s_d`, `d_d`, `pi` and `ni`, where `ni` is a set of characters such that if `ni` is nonempty then `d_d` is nonempty and `(car d_d)` $\notin$ `ni`, then evaluating

```
(matchorig p d (append pi p) (append pi d))
```

(where `d` stands for the concatenation of `s_d` and `d_d`), and evaluating

```
(matchneg p s_d ni d_d pi)
```

yield the same result.

Proof: The proof is by induction on the tuple $\langle|\texttt{d\_d}|, |\texttt{(append pi s\_d)}|, |\texttt{p}|\rangle$, ordered lexicographically. It is straightforward to show that this value decreases at every function call of match_{neg}. We thus do not explicitly check that the induction hypothesis applies in each case. As in Appendix 4.B, we use a more mathematical notation for conciseness.

We want to show that for all p, d_s, d_d, pi and ni, where ni is a set of characters such that if d_d is nonempty, then $\mathsf{hd}(d_d) \notin ni$,

$$\text{match}_{neg}(p, s_d, ni, d_d, pi) = \text{match}_{orig}(p, s_d@d_d, pi@p, pi@s_d@d_d).$$

In all cases where $p = [\,]$, the calls to match_{neg} and match_{orig} evaluate to *accept*. Otherwise, assume that the theorem is true for all smaller tuples. We proceed with an exhaustive case distinction. In the following, we assume for every case that none of the preceeding cases holds.

Consider the case that $s_d = [\,]$ and $ni = [\,]$:

- $d_d = [\,]$: In this case the call to match_{neg} evaluates to *reject*. If both s_d and d_d are empty, then the second argument of match_{orig} is empty as well, and the call to match_{orig} also evaluates to *reject*.

- $\mathsf{hd}(p) = \mathsf{hd}(d_d)$: In this case the call to match_{neg} evaluates to

$$\text{match}_{neg}(\mathsf{tl}(p), [\,], [\,], \mathsf{tl}(d_d), pi@[\mathsf{hd}(p)]),$$

  and the call to match_{orig} evaluates to

$$\text{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(d_d), pi\,, pi@d_d).$$

  Both are equal:

$$\begin{array}{ll} & \text{match}_{neg}(\mathsf{tl}(p), [\,], [\,], \mathsf{tl}(d_d), pi@[\mathsf{hd}(p)]) \\ \overset{\text{IH}}{=} & \text{match}_{orig}(\mathsf{tl}(p), [\,]@\mathsf{tl}(d_d), pi@[\mathsf{hd}(p)]@\mathsf{tl}(p), pi@[\mathsf{hd}(p)]@[\,]@\mathsf{tl}(d_d)) \\ = & \text{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(d_d), pi@p, pi@d_d) \end{array}$$

- $pi = [\,]$: In this case the call to match_{neg} evaluates to

$$\text{match}_{neg}(p, [\,], [\,], \mathsf{tl}(d_d), [\,]),$$

  and the call to match_{orig} evaluates to

$$\text{match}_{orig}(p, \mathsf{tl}(d_d), p, \mathsf{tl}(d_d)).$$

  Both are equal:

$$\begin{array}{ll} & \text{match}_{neg}(p, [\,], [\,], \mathsf{tl}(d_d), [\,]) \\ \overset{\text{IH}}{=} & \text{match}_{orig}(p, [\,]@\mathsf{tl}(d_d), [\,]@p, [\,]@[\,]@\mathsf{tl}(d_d)) \\ = & \text{match}_{orig}(p, \mathsf{tl}(d_d), p, \mathsf{tl}(d_d)) \end{array}$$

- $pi \neq [\,]$: In this case the call to match_{neg} evaluates to

$$\text{match}_{neg}(pi@p, \mathsf{tl}(pi\,), [\mathsf{hd}(p)], d_d, [\,]),$$

  and the call to match_{orig} evaluates to

$$\text{match}_{orig}(pi@p, \mathsf{tl}(pi@d_d), pi@p, \mathsf{tl}(pi@d_d)).$$

  Both are equal:

$$\begin{array}{ll} & \text{match}_{neg}(pi@p, \mathsf{tl}(pi\,), [\mathsf{hd}(p)], d_d, [\,]) \\ \overset{\text{IH}}{=} & \text{match}_{orig}(pi@p, \mathsf{tl}(pi\,)@d_d, [\,]@pi@p, [\,]@\mathsf{tl}(pi\,)@d_d) \\ = & \text{match}_{orig}(pi@p, \mathsf{tl}(pi@d_d), pi@p, \mathsf{tl}(pi@d_d)) \end{array}$$

Now we turn to the case that $s_d = [\,]$ and $ni \neq [\,]$. Recall that ni is a set of characters that does not contain the first element of d_d.

- $\mathsf{hd}(p) \in ni$ and $pi = [\,]$: In this case the call to match_{neg} evaluates to $\text{match}_{neg}(p, [\,], [\,], \mathsf{tl}(d_d), [\,])$. $\mathsf{hd}(p) \in ni$ implies that $\mathsf{hd}(p) \neq \mathsf{hd}(d_d)$, and since $s_d = [\,]$, $\mathsf{hd}(p) \neq \mathsf{hd}(s_d @ d_d)$. Thus the call to match_{orig} evaluates to $\text{match}_{orig}(p, \mathsf{tl}(d_d), p, \mathsf{tl}(d_d))$. Both are equal:

$$\begin{array}{ll} & \text{match}_{neg}(p, [\,], [\,], \mathsf{tl}(d_d), [\,]) \\ \overset{\text{IH}}{=} & \text{match}_{orig}(p, [\,]@\mathsf{tl}(d_d), [\,]@p, [\,]@[\,]@\mathsf{tl}(d_d)) \\ = & \text{match}_{orig}(p, \mathsf{tl}(d_d), p, \mathsf{tl}(d_d)) \end{array}$$

- $\mathsf{hd}(p) \in ni$ and $pi \neq [\,]$: In this case the call to match_{neg} evaluates to $\text{match}_{neg}(pi@p, \mathsf{tl}(pi), ni, d_d, [\,])$, and again the call to match_{orig} evaluates to $\text{match}_{orig}(pi@p, \mathsf{tl}(pi@d_d), pi@p, \mathsf{tl}(pi@d_d))$. Both are equal:

$$\begin{array}{ll} & \text{match}_{neg}(pi@p, \mathsf{tl}(pi), ni, d_d, [\,]) \\ \overset{\text{IH}}{=} & \text{match}_{orig}(pi@p, \mathsf{tl}(pi)@d_d, [\,]@pi@p, [\,]@\mathsf{tl}(pi)@d_d) \\ = & \text{match}_{orig}(pi@p, \mathsf{tl}(pi@d_d), pi@p, \mathsf{tl}(pi@d_d)) \end{array}$$

- $\mathsf{hd}(p) = \mathsf{hd}(d_d)$: In this case the call to match_{neg} evaluates to

$$\text{match}_{neg}(\mathsf{tl}(p), [\,], [\,], \mathsf{tl}(d_d), pi@[\mathsf{hd}(p)]),$$

and the call to match_{orig} evaluates to

$$\text{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(d_d), pi@p, pi@d_d).$$

Both are equal:

$$\begin{array}{ll} & \text{match}_{neg}(\mathsf{tl}(p), [\,], [\,], \mathsf{tl}(d_d), pi@[\mathsf{hd}(p)]) \\ \overset{\text{IH}}{=} & \text{match}_{orig}(\mathsf{tl}(p), [\,]@\mathsf{tl}(d_d), pi@[\mathsf{hd}(p)]@\mathsf{tl}(p), pi@[\mathsf{hd}(p)]@[\,]@\mathsf{tl}(d_d)) \\ = & \text{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(d_d), pi@p, pi@d_d) \end{array}$$

- $pi = [\,]$: In this case the call to match_{neg} evaluates to

$$\text{match}_{neg}(p, [\,], [\,], \mathsf{tl}(d_d), [\,]),$$

and the call to match_{orig} evaluates to

$$\text{match}_{orig}(p, \mathsf{tl}(d_d), p, \mathsf{tl}(d_d)).$$

Both are equal:

$$\begin{array}{ll} & \text{match}_{neg}(p, [\,], [\,], \mathsf{tl}(d_d), [\,]) \\ \overset{\text{IH}}{=} & \text{match}_{orig}(p, [\,]@\mathsf{tl}(d_d), [\,]@p, [\,]@[\,]@\mathsf{tl}(d_d)) \\ = & \text{match}_{orig}(p, \mathsf{tl}(d_d), p, \mathsf{tl}(d_d)) \end{array}$$

- $pi \neq [\,]$: In this case the call to match_{neg} evaluates to

  $$\text{match}_{neg}(pi@p, \mathsf{tl}(pi), \mathsf{hd}(p) :: ni, d_d, [\,]),$$

  and the call to match_{orig} evaluates to

  $$\text{match}_{orig}(pi@p, \mathsf{tl}(pi@d_d), pi@p, \mathsf{tl}(pi@d_d)).$$

  Both are equal:

  $$\begin{array}{ll} & \text{match}_{neg}(pi@p, \mathsf{tl}(pi), \mathsf{hd}(p) :: n, d_d, [\,]) \\ \overset{\text{IH}}{=} & \text{match}_{orig}(pi@p, \mathsf{tl}(pi)@d_d, [\,]@pi@p, [\,]@\mathsf{tl}(pi)@d_d) \\ = & \text{match}_{orig}(pi@p, \mathsf{tl}(pi@d_d), pi@p, \mathsf{tl}(pi@d_d)) \end{array}$$

Finally, we consider the case that $s_d \neq [\,]$:

- $\mathsf{hd}(p) = \mathsf{hd}(s_d)$: In this case the call to match_{neg} evaluates to

  $$\text{match}_{neg}(\mathsf{tl}(p), \mathsf{tl}(s_d), ni, d_d, pi@[\mathsf{hd}(p)]),$$

  and the call to match_{orig} evaluates to

  $$\text{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(s_d@d_d), pi@p, pi@s_d@d_d).$$

  Both are equal:

  $$\begin{array}{ll} & \text{match}_{neg}(\mathsf{tl}(p), \mathsf{tl}(s_d), ni, d_d, pi@[\mathsf{hd}(p)]) \\ \overset{\text{IH}}{=} & \text{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(s_d)@d_d, pi@[\mathsf{hd}(p)]@\mathsf{tl}(p), pi@[\mathsf{hd}(p)]@\mathsf{tl}(s_d)@d_d) \\ = & \text{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(s_d@d_d), pi@p, pi@s_d@d_d) \end{array}$$

- $\mathsf{hd}(p) \neq \mathsf{hd}(s_d)$: In this case the call to match_{neg} evaluates to

  $$\text{match}_{neg}(pi@p, \mathsf{tl}(pi@s_d), ni, d_d, [\,]),$$

  and the call to match_{orig} evaluates to

  $$\text{match}_{orig}(pi@p, \mathsf{tl}(pi@s_d@d_d), pi@p, \mathsf{tl}(pi@s_d@d_d)).$$

  Both are equal:

  $$\begin{array}{ll} & \text{match}_{neg}(pi@p, \mathsf{tl}(pi@s_d), ni, d_d, [\,]) \\ \overset{\text{IH}}{=} & \text{match}_{orig}(pi@p, \mathsf{tl}(pi@s_d)@d_d, [\,]@pi@p, [\,]@\mathsf{tl}(pi@s_d)@d_d) \\ = & \text{match}_{orig}(pi@p, \mathsf{tl}(pi@s_d@d_d), pi@p, \mathsf{tl}(pi@s_d@d_d)) \end{array}$$

□

4.D Pattern matching with positive and negative information: complexity of the specialized code

We present proofs of Theorems 4.6 and 4.7 about size and execution time of the specialized code for the matcher that uses positive and negative information (Figure 4.9). Figure 4.13 shows a binding-time annotated version of this matcher. Memoization points are indicated using comments. The subscripts on calls to `match` are used for referencing only.

4.D.1 Size

As in the proof of Theorem 4.1, we give an upper bound for the number of variants that are generated of each memoization point. We start by showing a lemma corresponding to Lemma 4.2.

Lemma 4.9 *In the evaluation of* `(main p0 d0)`*, for any* $k \geq 1$*, in the* k^{th} *call to* `match`*, the concatenation of argument* `pi` *with argument* `p` *is equal to* `p0`*.*

We can now calculate the number of variants of $M1$:

Lemma 4.10 *Specializing the implementation from Figure 4.9 with respect to a pattern of length* n *yields a residual program with at most* n *variants of* $M1$*.*

Proof: $M1$ is guarded by `s_d` $=$ `'()` $\wedge$ `ni` $=$ `'()`; thus, just as in the proof of Theorem 4.1, the number of variants is bounded by the number of different configurations of `p` and `pi`. Lemma 4.9 limits the number of different configurations of `p` and `pi` to $n+1$. Since $M1$ is never reached with `p` $=$ `'()`, we have an upper bound of n variants of $M1$. □

Next, we analyze the number of variants generated of $M2$. All variants of $M2$ arise from specializing the body of a variant of $M1$: the initial `s_d` and `ni` arguments of `match` are `'()`, so specialization of `main` first reaches either `'accept` or $M1$, rather than $M2$. We thus need to bound the number of variants of $M2$ generated in specializing the bodies of all variants of $M1$. In doing so, we assume that the names for the variants of $M1$ are generated in advance, so that specialization stops when an instance of $M1$ is reached. Under this assumption we can show the following lemma:

Lemma 4.11 *When specializing* `(match p s_d ni d_d pi)` *with respect to static input* `p`*,* `s_d`*,* `ni` *and* `pi` *such that specialization stops when an instance of* $M1$ *is encountered, then (1) if* `ni` $=$ `'()`*, no variant of* $M2$ *is generated; (2) if* `ni` $\neq$ `'()`*, then at most* $|A - \texttt{ni}|$ *variants of* $M2$ *are generated (where* A *is the set of characters contained in* `(append pi p)`*).*

```
(define (main p^S d^D)
  (match p '() '() d '()))

(define (match p^S s_d^S ni^S d_d^D pi^S)
  (cond
    [(null? p) 'accept]
    [(and (null? s_d) (null? ni))
     (cond                                          ; memoization point M1
       [(null? d_d) 'reject]
       [(equal? (car p) (car d_d))
        (match1 (cdr p) '() '() (cdr d_d)
                (append pi (list (car p))))]
       [else
        (if (null? pi)
            (match2 p '() '() (cdr d_d) '())
            (match3 (append pi p) (cdr pi) (list (car p))
                    d_d '()))])]
    [(null? s_d)
     (if (member (car p) ni)
         (if (null? pi)
             (match4 p '() '() (cdr d_d) '())
             (match5 (append pi p) (cdr pi) ni d_d '()))
         (if (equal? (car p) (car d_d))             ; memoization point M2
             (match6 (cdr p) '() '() (cdr d_d)
                     (append pi (list (car p))))
             (if (null? pi)
                 (match7 p '() '() (cdr d_d) '())
                 (match8 (append pi p) (cdr pi)
                         (cons (car p) ni) d_d '()))))]
    [else
     (cond
       [(equal? (car p) (car s_d))
        (match9 (cdr p) (cdr s_d) ni d_d (append pi (list (car p))))]
       [else
        (match10 (append pi p) (cdr (append pi s_d)) ni d_d '())])]))
```

Figure 4.13: The annotated string matcher (with negative information)

Proof: We begin with part (1): Because by assumption we start with `ni` being empty, and because $M2$ is guarded with `ni` $\neq$ `'()`, `ni` has to be augmented before $M2$ can be reached. The only call that adds an element to `ni` and is reachable with `ni` = `'()` is in the scope of $M1$. Therefore, by assumption, it is never reached, so no variant of $M2$ is generated.

Now we show part (2): We order configurations of static data $\langle \mathtt{p}, \mathtt{s\_d}, \mathtt{ni}, \mathtt{pi} \rangle$ according to the lexicographic order on $\langle |A - \mathtt{ni}|, |\mathtt{(append\ pi\ s\_d)}|, |\mathtt{p}| \rangle$; we prove by well-founded induction with respect to this ordering. Suppose `match`

is specialized with respect to the static input $\langle \texttt{p}, \texttt{s\_d}, \texttt{ni}, \texttt{pi} \rangle$. The proof is by an exhaustive case distinction. We consider only the case where $\texttt{s\_d} = \texttt{'()} \wedge \texttt{ni} \neq \texttt{'()} \wedge \texttt{(car p)} \notin \texttt{ni}$.

If $\texttt{s\_d} = \texttt{'()} \wedge \texttt{ni} \neq \texttt{'()} \wedge \texttt{(car p)} \notin \texttt{ni}$ then a variant of $M2$ is generated; we have to make sure that specializing with respect to all new calls that are encountered generates strictly fewer than $|A - \texttt{ni}|$ variants of $M2$. Calls 6, 7, and 8 are reachable. From the first part of this lemma we know that calls 6 and 7 produce no variants of $M2$, because they call `match` with $\texttt{ni} = \texttt{'()}$. For call 8 the induction hypothesis holds; because in the call the parameter `ni` is augmented by one character, at most $|A - \texttt{ni}| - 1$ variants of $M2$ are generated. So in all, at most $|A - \texttt{ni}|$ variants are generated. □

We are now in a position to prove Theorem 4.6.

Proof: Analyzing Figure 4.13 we see that null-tests are only generated at $M1$. Lemma 4.10 shows that at most n variants of $M1$ are generated.

Each of the at most n variants of $M1$ also generates one comparison in the residual code. Each variant of $M2$ also generates one such comparison. Hence it remains to show that $M2$ only gives rise to $n \cdot (c-1)$ variants. Consider the specialization of the body of one of the at most n variants of $M1$; the body contains the call sites 1, 2 and 3. From the first part of Lemma 4.11 we can infer that call 1 and call 2 never produce any variants of $M2$. For call 3, the second part of Lemma 4.11 tells us that at most $(c-1)$ variants of $M2$ are generated. Multiplying this by at most n variants of $M1$ gives us the desired upper bound of at most $n \cdot (c-1)$ variants of $M2$. Thus, in all, we have $n \cdot c$ residual equality tests. □

4.D.2 Execution time

When running the result of specializing a matcher with respect to `p` on some data string `d`, the number of equality and null tests performed is equal to the number of *irreducible* equality and null tests performed by the unspecialized matcher when applied to `p` and `d` (cf. the proof of Lemma 4.4). The following lemma gives an upper bound on the number of irreducible null tests performed by the optimized matcher from Figure 4.9 (referred to as $\texttt{main}_{\texttt{neg}}$ and $\texttt{match}_{\texttt{neg}}$ in this section). It further shows that the optimized matcher performs at most as many irreducible equality and null tests as the matcher that only uses positive information from Figure 4.5 ($\texttt{main}_{\texttt{pos}}$ and $\texttt{match}_{\texttt{pos}}$).

Lemma 4.12 *For all* `p`, `s_d`, `ni` *and* `d_d` *such that if* $\texttt{d\_d} = \texttt{'()}$ *then* $\texttt{ni} = \texttt{'()}$ *and* $\texttt{(car d\_d)} \notin \texttt{ni}$ *otherwise, (1) the number of irreducible null tests performed by* $(\texttt{match}_{\texttt{neg}}\ \texttt{p s\_d ni d\_d pi})$ *is at most* $|\texttt{d\_d}| + 1 - \mathit{sign}(|\texttt{ni}|)$ *(where* $\mathit{sign}(0) = 0$ *and* $\mathit{sign}(n) = 1$ *for* $n > 0$*), and (2)* $(\texttt{match}_{\texttt{neg}}\ \texttt{p s\_d ni d\_d pi})$ *always performs at most as many irreducible null tests and equality tests as* $(\texttt{match}_{\texttt{pos}}\ \texttt{p s\_d d\_d pi})$.

Proof: We prove by well-founded induction with respect to a termination relation for $\texttt{match}_{\texttt{neg}}$. The proof of (1) proceeds just like the proof of Lemma 4.4. The proof of (2) is by a straighforward exhaustive case distinction over the input. □

Theorem 4.7 follows immediately from Lemma 4.12.

Bibliography

[1] Maria Alpuente, Moreno Falaschi, Pascual Juliàn, and German Vidal. Specialization of inductively sequential functional logic programs. In Charles Consel, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 151–162, Amsterdam, The Netherlands, June 1997. ACM Press.

[2] Torben Amtoft. *Sharing of Computations*. PhD thesis, DAIMI, Department of Computer Science, University of Aarhus, 1993. Technical report PB-453.

[3] Torben Amtoft, Charles Consel, Olivier Danvy, and Karoline Malmkjær. The abstraction and instantiation of string-matching programs. Technical Report BRICS RS-01-12, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, April 2001.

[4] Anders Bondorf. Improving binding times without explicit cps-conversion. In *Proceedings of the Conference on Lisp and Functional Programming*, pages 1–10, San Francisco, CA, June 1992. ACM Press.

[5] Anders Bondorf. Similix 5.0 manual. Technical report, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1993. Included in the Similix 5.0 distribution.

[6] Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In David A. Schmidt, editor, *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, Copenhagen, Denmark, June 1993. ACM Press.

[7] Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, January 1989.

[8] Yoshihiko Futamura and Kenroku Nogi. Generalized partial computation. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 83–116. North Holland, 1988.

[9] Robert Glück and Jesper Jørgensen. Generating transformers for deforestation and supercompilation. In B. Le Charlier, editor, *Static Analysis*,

volume 864 of *Lecture Notes in Computer Science*, pages 432–448. Springer-Verlag, 1994.

[10] Robert Glück and Andrei Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In Patrick Cousot, Moreno Falaschi, Gilberto Filè, and Antoine Rauzy, editors, *Proceedings of the Third International Workshop on Static Analysis WSA'93*, number 724 in Lecture Notes in Computer Science, pages 112–123, Padova, Italy, September 1993. Springer-Verlag.

[11] Robert Glück and Valentin F. Turchin. Application of metasystem transition to function inversion and transformation. In *Proceedings of the international symposium on symbolic and algebraic computation*, pages 286–287, Tokyo, Japan, August 1990. ACM, ACM Press.

[12] Carsten K. Holst and Carsten K. Gomard. Partial evaluation is fuller laziness. In Hudak and Jones [13], pages 62–71.

[13] Paul Hudak and Neil D. Jones, editors. *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 26, No 9, New Haven, Connecticut, June 1991. ACM Press.

[14] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice Hall, June 1993.

[15] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[16] Jesper Jørgensen. Generating a pattern matching compiler by partial evaluation. In Simon L. Peyton Jones, Guy Hutton, and Carsten K. Holst, editors, *Functional Programming, Glasgow 1990*, Workshops in Computing, pages 177–195, Glasgow, Scotland, 1990. Springer-Verlag.

[17] Keiichi Kaneko and Masato Takeichi. Derivation of a Knuth-Morris-Pratt algorithm by fully lazy partial computation. *Advances in Software Science and Technology*, 5:11–24, 1993.

[18] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998. Available online at `http://www.brics.dk/~hosc/11-1/`.

[19] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[20] Laura Lafave and John P. Gallagher. Constraint-based partial evaluation of rewriting-based functional logic programs. In Norbert E. Fuchs, editor, *7th*

International Workshop on Program Synthesis and Transformation, number 1463 in Lecture Notes in Computer Science, pages 168–188, Leuven, Belgium, July 1997. Springer-Verlag.

[21] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 27–29, Orlando, FL, June 1994. ACM Press.

[22] Jonathan Martin and Michael Leuschel. Sonic partial deduction. In Dines Bjørner, Manfred Broy, and Alexander V. Zamulin, editors, *Perspectives of System Informatics, Third International Andrei Ershov Memorial Conference*, number 1755 in Lecture Notes in Computer Science, pages 101–112, Akademgorodok, Novosibirsk, Russia, July 1999. Springer-Verlag.

[23] Torben Æ. Mogensen. Separating binding times in language specifications. In *Fourth International Conference on Functional Programming and Computer Architecture (FPCA)*, pages 12–25, London, UK, September 1989. ACM Press.

[24] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science. Prentice-Hall International, 1987.

[25] Christian Queinnec and Jean-Marie Geffroy. Evaluation applied to symbolic pattern matching with intelligent backtrack. In Antoine Rauzy, editor, *Actes WSA'92 Workshop on Static Analysis*, volume 81-82 of *Series Bigre*, pages 109–117, Campus de Beaulieu, September 1992. Atelier Irisa, IRISA.

[26] Peter Sestoft. ML pattern match compilation and partial evaluation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 446–464, Dagstuhl, Germany, February 1996. Springer-Verlag.

[27] Donald A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In Hudak and Jones [13], pages 62–71.

[28] Morten H. Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6:811–838, 1996.

Chapter 5

Cost Recurrences for DML Programs

Abstract

A cost recurrence describes an upper bound for the running time of a program in terms of the size of its input. Finding cost recurrences is a frequent intermediate step in complexity analysis, and this step requires an abstraction from data to data size. In this article, we use information contained in dependent types to achieve such an abstraction: Dependent ML (DML), a conservative extension of ML, provides dependent types that can be used to associate data with size information, thus describing a possible abstraction. We systematically extract cost recurrences from first-order DML programs, guiding the abstraction from data to data size with information contained in DML type derivations.

5.1 Introduction

Analyzing the time complexity of a program is usually carried out in two steps. First, one establishes an upper bound of the program's running time as a function of the size of its input. Second, one approximates the growth of this extracted bounding function, thus determining the complexity class of the program. The first step requires an abstraction from data to data size. Information contained in *dependent types* can be used to achieve such an abstraction. In this article, we show how to automatically extract time bounds from first-order programs written in Dependent ML (DML), an extension of ML that provides a limited form of dependent types. If a bound is successfully extracted, we can guarantee that it is a *recurrence*, i.e., an equation defining a function in terms of its result on smaller inputs. A recurrence that describes an upper bound for the running time of a program is called a *cost recurrence*.

Limits and achievements of automated cost analysis *Automated* cost analyses have inherent limits. For example, finding a cost recurrence for a program is at least as hard as proving termination of the program. Also, finding good approximations for the growth of recurrences (or, in general, almost any kind of function) is known to be a hard problem.

Nevertheless, automated methods for cost analysis have been proposed. One choice is to restrict the class of programs such that termination is guaranteed, and the extracted time bounds can easily be approximated. For example, Reistad and Gifford [7] consider functional programs without general recursion, using only combinators such as `map` and `zip`. Methods that treat more general programs, as for example proposed by Le Métayer [4] and Rosendahl [8], usually focus on extracting a cost *program* p^c; if p^c terminates, it calculates an upper bound for the running time of a program p. Transforming p^c may yield a version compact enough to read off the complexity class of p. If not, p^c still may be useful for more empirical attempts to determine the time complexity of p, such as plotting input size against the running time calculated by p^c.

Dependent ML DML, which was developed by Xi [12, 16] in his PhD thesis, extends ML with a limited form of dependent types: A DML type can be *enriched* with indices taken from a constraint domain (e.g., integers equipped with their usual operations, with linear (in)equalities as constraints). For example, the data type of lists can be enriched with a notion of length or a data type of trees with a notion of height. The type language is expressive enough to encode well-formedness criteria, such as a tree being balanced. DML function types can express non-trivial properties, for example that a list is always mapped to a list of the same length, or that a function with a balanced tree as input always returns a balanced tree.

The design philosophy of DML is to use type-checking for the verification of non-trivial correctness properties of ML programs—every valid ML program is a valid DML program, because DML extends ML conservatively. For example, to verify that a program for inserting an element into a balanced tree indeed returns a balanced tree, the user needs to (1) enrich a data type of trees such that only balanced trees are accepted, and (2) declare in a type annotation that the insert function maps balanced trees to balanced trees. A range of similar examples convincingly demonstrates that DML is a useful tool for practical programming [12, 13, 15, 16].

This work We use information contained in DML type derivations to extract cost recurrences from DML programs. With DML types, data can be associated with a measure of data size, which essentially describes an abstraction from data to data size that is necessary for extracting a cost recurrence. For example, enriching the data type of lists with a notion of length describes an abstraction from lists to their length. More intricate measures—the high expressiveness of DML types allows the user to tailor measures to each situation. In many cases, measures with several components (e.g., for trees, the pair of height and number of leaves) prove to be useful. Size measures often coincide with shape information for data that is useful for verifying program properties by DML

type-checking. Therefore, in many cases, DML types that express correctness properties of a program can be reused for establishing the complexity of the program.

We allow recurrences to contain logical formulas, which are used to restrict arguments that cannot be completely determined. Thus, logical information contained in DML type derivations about such arguments can be included in cost recurrence rather than approximating them in an ad-hoc way. Compared with other methods that extract executable cost bounds—and therefore are required to carry out approximations—we leave the choice of how to approximate to the user, thus separating concerns between extracting a cost recurrence and solving it.

We combine the extraction of a cost bound with a check whether the result is indeed a recurrence: The information contained in DML type derivations facilitates a check of whether the size measure decreases for each recursive call under a wellfounded ordering. In other words, the user has to choose a size measure that constitutes a termination order for the program in question. This is no limitation compared to other methods: Because finding a cost bound entails a termination proof, in all methods for cost analysis a termination proof needs to be found in some way. It is an asset of using DML that the termination proof can be concisely encoded through the size measure.

An example Consider a `zip` function written in ML:

```
fun zip lp =
 case lp of
    (nil,nil) => nil
  | (cons(x,xs),cons(y,ys)) => cons((x,y),zip(xs,ys))
```

DML offers the possibility to annotate `zip` with a type containing an enriched version of lists. We enrich the data type of lists with a notion of length; the type of α-typed lists consisting of n elements is written as $\alpha\,\mathsf{list}(n)$. Obviously, `zip` should take two lists of equal length and return a list of the same length. DML type checking validates that `zip` has the type

$$\Pi\, n : \mathbb{N}\,.\, \alpha\,\mathsf{list}(n) \times \beta\,\mathsf{list}(n) \to (\alpha \times \beta)\mathsf{list}(n).$$

Intuitively, Π can be read as "for all".[1] In ML, a pair of two lists with different lengths could be passed to `zip`, which would result in a runtime error. In DML, the given type of `zip` allows `zip` only to be called with two lists of equal length. The type also shows that the resulting list is of the same length as the input lists.

Let us measure running time as the number of calls to user-defined functions, giving each call a cost of one unit. The resulting recurrence describes the number of calls to `zip` as a function of the length of the two input lists:

[1] Formally, Π introduces a dependent product, i.e., a product where the value of the first component (here n) determines the type of the second component (here the function from a pair of lists of length n to a list of length n). Dependent products are also called Π-types.

$$zip^c(n) = \begin{cases} n = 0 \to 0 \\ n > 0 \to 1 + zip^c(n-1) \end{cases}$$

This cost recurrence is extracted from a DML type derivation for `zip`. In the type derivation, the arguments to `zip`—two lists—are associated with an index n that represents their length. Using this information, the extraction algorithm abstracts from the lists to their length n. For example, the case expression is turned into a conditional by inferring for each branch a condition on n that has to hold if the pattern is matched. Similarly, the algorithm derives from the type derivation that the recursive call of `zip` has a list of length $n-1$ as argument, and thus generates a call $zip^c(n-1)$. Obviously, $n-1 < n$, so the extracted bound is a recurrence.

The recurrence can easily be solved: $zip^c(n) = n$.

The remainder of the article

The article is structured as follows: Section 5.2 gives an introduction to DML, Section 5.3 presents an intuitive account of our method for extracting cost recurrences and gives several examples, Section 5.4 contains a formal account, Section 5.5 treats related work, and Section 5.6 concludes. Appendix 5.A gives a short overview over the formal definition of DML, and Appendix 5.B contains details of the formal development of this work.

5.2 Background: Dependent ML

DML provides dependent types in which type index objects are limited to some constraint domain C. Type checking for DML is decidable; it is based on solving constraints in C. For dependently typed languages with significantly more expressive types (e.g., Cayenne [1]) type checking is undecidable.

We consider an effect-free fragment of DML. As constraint domain, we choose integers, constrained by linear (in)equalities—we write $\mathbb{Z}$ both for the sort of integers and the constraint domain. In the following, we present a short introduction to programming in DML and sketch the formal specification of DML. The latter forms the basis for the formal development presented in Section 5.4.

5.2.1 A programmer's view of DML

The only new aspect for an ML programmer is the extended type system, which contains type indices, in the present case integers.

Enriched recursive data types

As indicated in the example in Section 5.1, in DML a list type can be enriched with a notion of length, enabling us to express the type of α-typed lists of n elements as $\alpha\,\mathsf{list}(n)$. The DML data-type definition is

```
datatype α list with ℕ =
   nil(0)
 | Π n : ℕ . cons(n + 1) of α × α list(n)
```

It is obtained from an ordinary definition of lists in ML by making the following additions:

1. The phrase "`with` ℕ" has been added. This signifies that the data type of lists is to be enriched with one index and that this index is restricted to the sort of natural numbers. The constraint language of DML allows the definition of subsorts of an already defined sort: ℕ stands for $\{k : \mathbb{Z} \mid k \geq 0\}$.

2. Constructors and occurrences of "`list`" are augmented with an index. The constructor `nil` is indexed with 0, thus defining the empty list to be of type $\alpha\,\mathsf{list}(0)$. A list built with `cons` is of type $\alpha\,\mathsf{list}(n+1)$, provided that `cons` was applied to an element of type $\alpha\,\mathsf{list}(n)$. Hence, `cons` is indexed with $n + 1$.

3. The definition of the `cons` case exhibits a quantification over an index variable n. This index variable is necessary to express the dependence between the index of `cons` and the index of the list appearing in its branch. The quantification restricts the index variable to the sort of natural numbers.

Similarly, we can define the data type of a list of lists $\alpha\,\mathsf{llist}(m,n)$ as

```
datatype α llist with (ℕ,ℕ) =
   lnil(0,0)
 | Π m, n1, n2 : ℕ . lcons(m + 1,n1 + n2) of
                     α list(n1) × α llist(m,n2)
```

The first index stands for the number of inner lists and the second index for the total number of elements the inner lists contain.

The example of lists provides some intuition of how to define enriched recursive data types in two steps: First, decide on the number of indices to be used in the data type, along with the sorts the indices are to be restricted to. Second, annotate each constructor with the appropriate indices. When an index of a constructor depends on indices of recursive data types that appear under that constructor, introduce new index variables using quantification. An index can be defined as a function of other indices using all operations of the constraint domain.

In the introduction we mentioned that enriched data types can encode well-formedness criteria. As an example, we define a data type of height-balanced trees, i.e., the height difference between the two children of a node can be at most one:

```
datatype α HBtree with (ℕ,ℕ) =
   Leaf(0,0)
 | Π s1, s2, h1 : ℕ . Π h2 : {k : ℕ | h1 − 1 ≤ k ≤ h1 + 1} .
    Node(1 + max(h1, h2), s1 + s2 + 1) of
     α HBtree(h1,s1) × α × α HBtree(h2,s2)
```

We use two indices, where the first represents the height of the tree and the second represents the number of elements stored in the tree. When defining a node, we require for the heights h_1 and h_2 of the subtrees, that they differ by at most one. This can be achieved by (1) defining a sort of natural numbers k that differ by at most one from h_1, and (2) restricting h_2 to this new sort. As a consequence, only two trees with a height difference of at most one can be the children of a node, i.e., only height-balanced trees can have type $\alpha\,\mathsf{HBtree}$.

Note that for defining a new sort, all predicates and operations of the chosen constraint domain can be used. In the case of height-balanced trees, we use $-$, $+$, and $\leq$.

DML function types

As in ML, a data-type definition gives rise to type declarations for its constructors. For example, the definition of enriched lists presented above yields a type $\alpha\,\mathsf{list}(0)$ for the constructor `nil`, and the type

$$\Pi\, n : \mathbb{N}\,.\,\alpha \times \alpha\,\mathsf{list}(n) \to \alpha\ \mathsf{list}(n+1)$$

for the constructor `cons`.

```
append : Π n1,n2 : ℕ . α list(n1) × α list(n2) → α list(n1 + n2)
fun append lp =
 case lp of
    (nil,l2) => l2
  | (cons(x,xs),l2) => cons(x,append(xs,l2))


flatten : Π m,n : ℕ . α llist(m,n) → α list(n)
fun flatten ll =
 case ll of
    lnil => nil
  | lcons(xs,rest) => append(xs,flatten rest)

occurs : Π h, s : ℕ . string × string HBtree(h,s) → bool
fun occurs(e,t) =
 case t of
    Leaf => false
  | Node(t1,e',t2) => if e = e'
                      then true
                      else if e < e'
                           then occurs(s,t1)
                           else occurs(s,t2)
```

Figure 5.1: Some functions with their DML types

Figure 5.1 shows three functions operating on the data types defined in Section 5.2.1, together with their DML types:

- `append` takes two lists of n_1 and n_2 elements, respectively, and returns a list of $n_1 + n_2$ elements.
- `flatten` takes a list of lists that contain a total number of n elements, and returns a list of n elements
- `occurs` takes a string and a balanced tree, and returns a truth value, according to whether the string is stored in the tree or not (assuming a sorted balanced tree).

The DML types of `append`, `flatten` and `occurs` add shape information to the respective ML type of each function: In the case of `append` and `flatten`, we learn about the shape of the result, i.e., how long the output list is. For `occurs`, the DML type restricts the input tree to trees of a special shape, namely balanced trees.

So far, the output indices in the DML type of a function could always be specified as a function of the input indices. For relational dependencies, DML offers existential types. These allow one to restrict the index of an output to a sort—because sorts can be defined in terms of already declared indices, relational dependencies can be expressed.

Consider, for example, a function that inserts a string into a balanced tree. Depending on how the tree is rebalanced, the result can be a tree of equal height or a tree higher by one. Similarly, the number of elements in the tree stays equal if the element to be inserted already was in the tree, otherwise the number is increased by one. A valid DML type for a correct `insert` function on height-balanced trees is as follows:

$$\begin{aligned}\Pi\, h, s : \mathbb{N}\,.\, &\mathsf{string} \times \mathsf{HBtree}(h,s) \rightarrow \\ &\exists\, h' : \{k : \mathbb{N} \mid h \leq k \leq h+1\}\,. \\ &\exists\, s' : \{k : \mathbb{N} \mid s \leq k \leq s+1\}\,. \\ &\mathsf{HBtree}(h',s')\end{aligned}$$

The type of the output tree restricts height and size to be either equal or larger by one than the height and size of the input tree, respectively.

5.2.2 A formal specification of DML

In the theoretical development of DML [12], three languages are defined, whose interplay is displayed graphically in Figure 5.2.

- ML_0 basically is an extension of Mini-ML with general pattern matching. It formalizes a manageable subset of ML.
- $\mathrm{DML}_0^{\Pi}(C)$ is an explicitly typed language with dependent types, i.e., types that are indexed with elements from a constraint domain C. Its syntax is

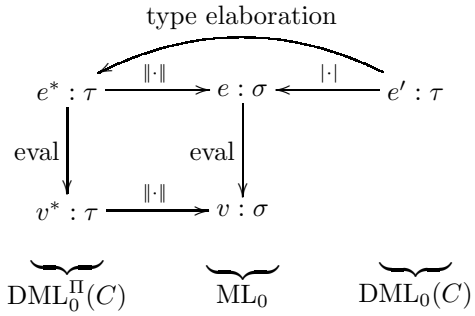

Figure 5.2: Interplay of languages

that of ML_0, adding abstraction over index variables and application of an expression to index expressions. A canonical erasure $\| \cdot \|$ that removes index-related syntax both from the term and the type language, maps $\mathrm{DML}_0^{\Pi}(C)$ into ML_0. The erasure commutes with evaluation.

- $\mathrm{DML}_0(C)$ enriches ML_0 with dependent types; it has the same type language as $\mathrm{DML}_0^{\Pi}(C)$. $\mathrm{DML}_0(C)$ requires type-annotations only for recursive definitions. An erasure on the type language extends to an erasure $| \cdot |$ that maps $\mathrm{DML}_0(C)$ into ML_0. A type-elaboration algorithm maps a $\mathrm{DML}_0(C)$ program with correct type annotations into $\mathrm{DML}_0^{\Pi}(C)$ such that (1) the type annotations are preserved and (2) the erasure of both terms results in the same ML_0 term.

$\mathrm{DML}_0^{\Pi}(C)$ allows easy type-checking, because it is explicitly typed and indices are part of the term language. For the same reason, however, $\mathrm{DML}_0^{\Pi}(C)$ is impractical for actual programming. Instead, the user works with $\mathrm{DML}_0(C)$, which corresponds to the language, the example programs of Section 5.2.1 are given in: Their displayed DML-types are the type-annotations that are required for the implicit recursive definitions. Type-checking is carried out by a type elaboration algorithm [12, Chapter 4], evaluation by applying the erasure and the ML_0 evaluation mechanism.

In the following, we first give some basic facts about constraint domains and the constraint language used to express the index objects for DML types. We then briefly describe $\mathrm{DML}_0^{\Pi}(C)$.[2] The description glosses over many details—we refer the reader to Xi's PhD thesis [12] for the complete picture.

[2]For simplicity, we restrict the presentation to the monomorphic case without existential types—polymorphism and existential types are treated in extensions of $\mathrm{DML}_0^{\Pi}(C)$.

Constraints in DML

A constraint domain C is defined by (1) a signature Σ that declares a base sort along with basic operations and predicates and (2) a Σ-structure. For example, for the constraint domain $\mathbb{Z}$, the signature declares the base sort $\mathbb{Z}$ and the usual operations ($+$, $-$, *mod*, etc.) and predicates ($<$, $\geq$, etc.) over integers; the Σ-structure is given by the standard model of integers.

$$
\begin{array}{lrcl}
\text{sorts} & \gamma & ::= & b \mid \mathbf{1} \mid \gamma_1 \times \gamma_2 \mid \{a : \gamma \mid P\} \\
\text{propositions} & P & ::= & \top \mid \bot \mid i = j \mid p(i) \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \\
\text{objects} & i, j & ::= & a \mid f(i) \mid \langle\rangle \mid \langle i, j\rangle \mid \mathbf{fst}(i) \mid \mathbf{snd}(i) \\
\text{contexts} & \phi & ::= & \cdot \mid \phi, a : \gamma \mid \phi, P
\end{array}
$$

Figure 5.3: Constraint language

DML uses the constraint language defined in Figure 5.3 to express the index objects for DML types. New sorts can be defined by pairing already defined sorts or restricting an already defined sort with a sort proposition. Sort propositions are built from the basic predicates p of the constraint domain. Index sorts serve as types for index objects, in which basic operations f of the constraint domain can appear. An index context is given as a collection of index propositions and type declarations for index variables.

DML type-checking requires a constraint solver that is able to handle constraints of the form

$$\Phi \quad ::= \quad \top \mid \bot \mid i = j \mid p(i) \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \forall a : \gamma.\Phi \mid \exists a : \gamma.\Phi$$

Constraint satisfaction under a given index context, which is written as $\phi \models \Phi$, is defined in the canonical way.

The language $\mathbf{DML}_0^{\Pi}(C)$

A grammar of the $\mathrm{DML}_0^{\Pi}(C)$ syntax is given in Figure 5.4.

$$
\begin{array}{rcl}
\tau & ::= & \delta(i) \mid \mathbf{1} \mid (\tau_1 \times \tau_2) \mid (\tau_1 \to \tau_2) \mid \Pi\, a : \gamma\,.\,\tau \\
e & ::= & x \mid \langle\rangle \mid \langle e_1, e_2\rangle \mid c[i_1]\ldots[i_n] \mid c[i_1]\ldots[i_n](e) \\
 & & \mid (\mathbf{case}\ e\ \mathbf{of}\ ms) \mid (\mathbf{lam}\ x : \tau\,.\,e) \mid e_1(e_2) \\
 & & \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} \mid (\mathbf{fix}\ x : \tau.e) \\
 & & \mid (\lambda a : \gamma\,.\,e) \mid e[i] \\
p & ::= & x \mid c[a_1]\ldots[a_n] \mid c[a_1]\ldots[a_n](p) \mid \langle\rangle \mid \langle p_1, p_2\rangle \\
ms & ::= & p \Rightarrow e \mid p \Rightarrow e \mid ms
\end{array}
$$

Figure 5.4: Syntax of $\mathrm{DML}_0^{\Pi}(C)$

In the grammar of the type language, $\delta(i)$ stands for a data type δ that is indexed with index object i. The DML data-type declaration of an enriched data type $\delta(i)$ yields constructor types of form $\Pi\, a_1 : \gamma_1 \ldots \Pi\, a_k : \gamma_k \,.\, \delta(i)$ for constructors without argument such as *nil*, and $\Pi\, a_1 : \gamma_1 \ldots \Pi\, a_k : \gamma_k \,.\, \tau \to \delta(i)$ for constructors with argument such as *cons*. Several examples of types appeared in Section 5.2.1.

In addition to the usual constructs, the term language provides abstraction over index variables ($\lambda a : \gamma \,.\, e$) and application of an expression to an index object ($e[i]$). Furthermore, a constructor c of a recursive data type only appears with a number of index arguments—index variables when appearing in a pattern p and index objects otherwise. The number and sorts of index arguments is determined by the constructor type, which is inferred from the corresponding data type definition (see Section 5.2.1).

A typing judgment for $\mathrm{DML}_0^{\Pi}(C)$ has the form

$$\phi; \Gamma \vdash e : \tau,$$

where ϕ is an index context and Γ a normal context; an overview over the typing rules for $\mathrm{DML}_0^{\Pi}(C)$ is presented in Appendix 5.A.1.

Substitutions play a central role in the formalization of DML: They are used both in the definition of the typing system and the semantics (Appendix 5.A.2). A substitution can both affect index variables and normal variables:

$$\theta ::= [] \mid \theta[a \mapsto i] \mid \theta[x \mapsto e]$$

For a substitution θ, its restriction to index variables is referred to as θ_ϕ, its restriction to normal variables as θ_Γ.

The application of a substitution θ to a term t is written $t[\theta]$. With $\theta_1 \circ \theta_2$ we denote the substitution mapping t to $(t[\theta_1])[\theta_2]$; with $\theta_1\theta_2$ we denote the substitution that behaves like θ_1 on all variables in $\mathbf{dom}(\theta_1)\backslash\mathbf{dom}(\theta_2)$, and like θ_2 on all other variables.

5.3 Extracting cost recurrences

We first give an intuitive account of our method for extracting cost recurrences from DML programs, deferring a formal treatment to Section 5.4. We then present examples illustrating some distinctive features of the method.

5.3.1 The intuition behind extracting cost recurrences

We extract cost recurrences from first-order DML programs of the form

$\mathtt{F_1} : \Pi\, a_0 : \gamma_0 \ldots \Pi\, a_{j_1} : \gamma_{j_1} \,.\, (\rho_{10} \times \rho_{11} \cdots \times \rho_{1l_1}) \to \rho_1$
`fun` $\mathtt{F_1(x_0, x_1, \ldots, x_{1_1})}$ `=` $\mathtt{e_1}$
$\vdots$
$\mathtt{F_k} : \Pi\, a_0 : \gamma_0 \ldots \Pi\, a_{j_k} : \gamma_{j_k} \,.\, (\rho_{k0} \times \rho_{k1} \cdots \times \rho_{kl_k}) \to \rho_k$
`fun` $\mathtt{F_k(x_0, x_1, \ldots, x_{1_k})}$ `=` $\mathtt{e_k}$

where we write $\rho ::= \delta(i) \mid \mathbf{1} \mid (\rho_1 \times \rho_2)$ for first-order types; also data-type constructors are only allowed to take first-order arguments. Because indices are used to abstract from data to data size, we require that (1) all sorts γ have been constructed only with subsorts of $\mathbb{N}$ and (2) data types are enriched such that for any i, all indices appearing in a branch of a data type $\delta(i)$ must be bounded.[3]

We count cost in terms of the number of calls to user-defined functions F and to constructors c, assigning a cost of $\mathbf{c}_F$ and $\mathbf{c}_c$ for each call, respectively. $\mathbf{c}_F$ and $\mathbf{c}_c$ are constants of the domain in which cost is measured, e.g., the natural numbers.

The first step of extracting a cost recurrence from a DML program is type elaboration, which yields a $\mathrm{DML}_0^{\Pi}(\mathbb{Z})$ program.[4]

Example *For the* `append` *function from Figure 5.1, type elaboration yields the* $DML_0^{\Pi}(\mathbb{Z})$ *program displayed in Figure 5.5. Type elaboration makes the indices explicit in the term language: index variables* n_1 *and* n_2 *are abstracted over; pattern matching against cons introduces a new index variable* n_1'*; cons and the recursive call of append are passed index objects that describe the length of the respective list arguments passed to cons and append. Because* $DML_0^{\Pi}(\mathbb{Z})$ *is monomorphic, assume that the data type* list *has been defined for a fixed type of elements, say* string*. The constructors nil and cons then are typed as follows:*

$$\begin{array}{lcl} \mathit{nil} & : & \mathsf{list}(0) \\ \mathit{cons} & : & \Pi\, n : \mathbb{N}\,.\, \mathsf{string} \times \mathsf{list}(n) \to \mathsf{list}(n+1) \end{array}$$

$$\begin{array}{l} \mathbf{fix}\ \mathit{append} : \Pi\, n_1 : \mathbb{N}\,.\, \Pi\, n_2 : \mathbb{N}\,.\, \mathsf{list}(n_1) \times \mathsf{list}(n_2) \\ \qquad\qquad\qquad\qquad\qquad\qquad \to \mathsf{list}(n_1 + n_2). \\ \quad \lambda n_1 : \mathbb{N}\,.\, \lambda n_2 : \mathbb{N}\,.\, \mathbf{lam}\ \mathit{lp} : \mathsf{list}(n_1) \times \mathsf{list}(n_2)\,. \\ \qquad \mathbf{case}\ \mathit{lp}\ \mathbf{of} \\ \qquad\quad \langle \mathit{nil}, l_2 \rangle \Rightarrow l_2 \\ \qquad\quad \mid \langle \mathit{cons}[n_1']\langle x, xs\rangle, l_2\rangle \Rightarrow \\ \qquad\qquad \mathit{cons}[n_1' + n_2]\langle x, \mathit{append}[n_1'][n_2]\langle xs, l_2\rangle\rangle \end{array}$$

Figure 5.5: The `append` function in $\mathrm{DML}_0^{\Pi}(C)$

[3] More precisely speaking, every constructor type $\Pi\, a_1 : \gamma_1 \ldots \Pi\, a_k : \gamma_k\,.\, \rho \to \delta(i)$ must be such that for a fixed $\delta(i)$, there are only finitely many $z_1, \ldots z_k$ such that $c[z_1]\ldots[z_k]$ is of type $\rho \to \delta(i)$. This condition is met for every data-type definition in which the indices convey size information: structures of a given size cannot contain substructures of arbitrary size.

[4] Because type elaboration as defined by Xi [12, Chapter 4] works on $\mathrm{DML}_0(C)$ programs, the program has to be rewritten in $\mathrm{DML}_0(\mathbb{Z})$. This is easily done by replacing the ML function-definition syntax with a recursive definition (keyword **fix**), a lambda-abstraction (keyword **lam**), and a case expression with a single pattern, and declaring $F_1 \ldots F_k$ in a row of nested let-statements.

We now describe intuitively how the extraction algorithm works. Note that all steps can be carried out automatically; for manipulating constraints, the algorithm uses a constraint solver for $\mathbb{Z}$.

The type of each function determines the arguments of the corresponding recurrence equation. A function

$$F : \Pi\, a_0 : \gamma_0 \ldots \Pi\, a_l : \gamma_l \,.\, \rho_1 \to \rho_2$$

gives rise to a recurrence equation F^c with $a_0 \ldots a_l$ as formal parameters.

Example (cont.) *For append, a recurrence equation append^c with formal parameters n_1 and n_2 is extracted.*

The extraction algorithm works on the body of the function definitions. The issues that have to be dealt with are

1. How to treat case expressions?
2. How to treat index variables introduced by pattern matching?
3. How to assign and add up cost?

How to treat case expressions? To abstract from data to data size, we need to turn case expressions, which examine data, into conditionals that examine data size. Such a transformation can be achieved using information contained in the $\mathrm{DML}_0^{\Pi}(\mathbb{Z})$ type derivation: During type checking, constraints over the index objects in the program are collected in an index context. Consider a branch of a case expression over some type ρ. The type derivation contains a collection of constraints that have to be satisfied when entering the branch, i.e., when the pattern of the branch is matched. By projecting out these constraints over the index variables contained in ρ, i.e., eliminating all other index variables, a guard for the corresponding branch of a conditional can be derived.

Example (cont.) *The case expression in append is type-checked under the index context*

$$\phi = n_1 : \mathbb{N}, n_2 : \mathbb{N}$$

For the two branches, additional constraints are generated:

- *For the branch with pattern $\langle nil, l_2 \rangle$, the index context $n_1 = 0$ is generated.*
- *For the branch with pattern $\langle cons[n_1']\langle x, xs\rangle, l_2\rangle$, the index context $n_1' : \mathbb{N}, n_1 = n_1' + 1$ is generated.*

From the conjunction of ϕ and the newly generated index context of each branch, we can derive a condition in terms of n_1 and n_2 by projecting out over n_1 and n_2: The result is $n_1 = 0$ for the first branch and $n_1 > 0$ for the second branch.

In general, it is possible that the generated guards overlap, even though the patterns of the case expression are mutually exclusive. When, during the evaluation of a recurrence equation, more than one guard is satisfied, all possible branches are evaluated and the maximum value is returned.

How to treat index variables introduced by pattern matching? A pattern can introduce new index variables; these index variables may appear within the branch guarded by the pattern, and thus also may play a role in the corresponding conditional branch of the extracted recurrence equation. Often, we can eliminate such index variables by deriving equality constraints that define a new index variable in terms of other index variables. If not, then the constraints can be used to derive a restriction for the values of the new index variables. This restriction is inserted into the extracted conditional branch.

Example (cont.) *The second branch of the case expression in append introduces the new index variable n_1'. The constraints allow us to derive that $n_1' = n_1 - 1$, so n_1' can be eliminated.*

How to count and add up cost? When extracting a cost recurrence, we need to count every call to a user-defined function F with a cost of $\mathbf{c}_F$ and every use of a constructor c with a cost of $\mathbf{c}_c$. Consider first a constructor c without arguments: In the cost recurrence, we simply replace $c[i_1]\dots[i_k]$ with $\mathbf{c}_c$. For constructors with arguments $c[i_1]\dots[i_k](e)$ and function calls $F[i_1]\dots[i_k](e)$, the cost incurred by e also needs to be taken into account. Hence, we first extract a recurrence-equation expression t that represents the cost of evaluating the argument, and then add it to the cost incurred by the function call:

- The total cost of $c[i_1]\dots[i_k](e)$ is $t + \mathbf{c}_c$
- The total cost of $F[i_1]\dots[i_k](e)$ is $t + \mathbf{c}_F + (F^c i_1 \dots i_k)$, where $F^c i_1 \dots i_k$ is a call to the cost recurrence extracted for F.

In our cost model, constants and variables can be accessed without cost, and therefore are turned into the constant 0 when extracting a cost recurrence.

Example (ended) *We now assemble all the pieces of a cost recurrence for append. If we assign a cost of one unit to recursive calls of append and assume the use of cons to be cost free, then the cost of append is described by*

$$append^c\ n_1\ n_2 = \begin{cases} n_1 = 0 \to 0 \\ n_1 > 0 \to 1 + append^c(n_1 - 1)\ n_2 \end{cases}$$

(In the second branch, we have removed additions of zero that resulted from the variables x, xs and l_2, and the application of constructor cons.)

5.3.2 Example: Flattening a list of lists

The `flatten` function (see Figure 5.1) is an interesting problem for extracting a cost recurrence because of the choice of size measure for the input: The size

of a list of lists is measured both in terms of the number of inner lists and the total number of elements contained in the inner lists.

We measure cost in terms of calls to user-defined functions. Our method yields the following cost recurrence:[5]

$$\mathit{flatten}^c\ m\ n = \begin{cases} m = 0 \wedge n = 0 \rightarrow 0 \\ m > 0 \rightarrow 2 + \mathit{append}^c\ n_1\ n_2 \\ \qquad\qquad + \mathit{flatten}^c\ (m-1)\ n_2 \end{cases} \quad \text{where} \quad n_1 + n_2 = n$$

Here, a restriction $n_1 + n_2 = n$ for the new variables n_1 and n_2 introduced by pattern matching has been inserted by the extraction algorithm—neither n_1 nor n_2 can be eliminated automatically.

Using the equation $\mathit{append}^c\ n_1\ n_2 = n_1$ derived in Section 5.3.1, we can rewrite the cost recurrence for *flatten* as

$$\mathit{flatten}^c\ m\ n = \begin{cases} m = 0 \wedge n = 0 \rightarrow 0 \\ m > 0 \rightarrow 2 + n_1 \\ \qquad\qquad + \mathit{flatten}^c\ (m-1)\ n_2 \end{cases} \quad \text{where} \quad n_1 + n_2 = n$$

It is easy to see that the maximal cost incurred by n_1 in the second branch, added over all recursive calls, is n; all in all, we can approximate the cost of `flatten` as $\mathit{flatten}^c(m, n) = 2m + n$.

The size measure chosen here for a list of lists is intuitive and seems to be crucial for deriving a useful bound. Yet it is unclear how this measure could be be defined without the expressiveness offered by DML types, e.g., when using abstract-interpretation techniques [8].

5.3.3 Example: Searching a balanced tree

The `occurs` function (see Figure 5.1) provides an example of how two cost bounds in terms of different size measures can be obtained: one in terms of the height of a tree, and one in terms of the number of elements stored in a tree. The latter bound is obtained by reasoning with DML type guarantees—we profit from the fact that DML can express properties of the input that are not inferable from the code.

[5] Here and in all the following recurrences we have simplified additions of constants.

Our method yields the following cost recurrence for `occurs`:

$$occurs^c\ h\ s = \begin{cases} h = 0 \wedge s = 0 \rightarrow 0 \\ h > 0 \wedge s > 0 \rightarrow \begin{cases} 0 \\ \begin{cases} 1 + occurs^c\ h_1\ s_1 \\ 1 + occurs^c\ h_2\ s_2 \end{cases} \end{cases} \end{cases}$$

$$\text{where} \quad \begin{array}{l} h_1 - 1 \leq h_2 \leq h_1 + 1 \\ \wedge\ max(h_1, h_2) + 1 = h \\ \wedge\ s_1 + s_2 + 1 = s \end{array}$$

(Each if expression gives rise to a guardless conditional, because no restrictions on indices can be inferred from its test expression.)

The recurrence looks more daunting than it is: It keeps track both of the height and the number of elements in the tree, but it is easy to see that the number of elements is of no consequence to the result of the cost recurrence. Approximating both h_1 and h_2 with $h - 1$ gives rise to a simple recurrence equation whose solution is $occurs^c(h, s) = h$.

The complication of eliminating size information could have been avoided by chosing a tree type which only keeps track of the height of a tree. Also keeping track of the number of elements, however, allows us to derive a cost measure in terms of the number of elements rather than the height of the tree. The crucial observation to make is that DML data-type definitions give rise to induction principles for proving relations among the indices of a data type. The definition of HBtree, for example, yields the following induction schema:

> For $R \in \mathbb{N} \times \mathbb{N}$, if
>
> 1. $R(0, 0)$
> 2. if for all h_1, h_2, s_1, s_2 with $h_1 - 1 \leq h_2 \leq h_1 + 1$, $R(h_1, s_1)$ and $R(h_2, s_2)$ it follows that $R(max(h_1, h_2) + 1, s_1 + s_2 + 1)$
>
> then whenever a value has type HBtree(h,s), the relation $R(h, s)$ holds.

Using this induction schema, one can show that $s \geq 2^h - 1$ for any height-balanced tree with height h and size s. Taking the logarithm, we see that $h \leq log(s + 1)$; combining this with the cost recurrence $occurs^c(h, s) = h$, we derive that the cost of `occurs` is logarithmic in s. This derivation is fully formal, i.e., based only on assumptions explicit in the types or the extracted recurrence.

5.3.4 Example: Merge sort

Merge sort provides an example of how the extraction algorithm preserves useful information contained in a program's DML type.

An implementation of merge sort in DML is given in Figure 5.6 (adapted from the distribution of *de Caml*, a DML prototype [11]): Function `initlist`

```
merge : Π n1, n2 : ℕ . list(n1) × list(n2) → list(n1 + n2)
fun merge lp =
 case lp of
    (nil, l2) => l2
  | (l1, nil) => l1
  | (cons(h1,t1),cons(h2,t2)) =>
      if h1 < h2 then cons(h1,merge(t1,l2))
                 else cons(h2,merge(l1,t2))

initlist : Π n : ℕ . list(n) → llist(⌈n/2⌉,n)
fun initlist l =
 case l of
    nil => lnil
  | cons(_,nil) => lcons(l, lnil)
  | cons(e1,cons(e2, rest)) =>
      lcons(if e1 < e2
            then cons(e1,cons(e2,nil))
            else cons(e2,cons(e1,nil)),
            initlist rest)

merge2 : Π m, n : ℕ . llist(m,n) → llist(⌈m/2⌉,n)
fun merge2 ll =
 case ll of
    lnil => ll
  | lcons(_,lnil) => ll
  | lcons(l1,lcons(l2,rest)) =>
       lcons(merge(l1,l2),merge2 rest)



mergeall : Π m, n : ℕ . llist(m,n) → list(n)
mergeall ll =
 case ll of
    lnil => nil
  | lcons(l,lnil) => l
  | lcons(_,lcons(_,_)) => mergeall(merge2 ll)

msort : Π n : ℕ . list(n) → list(n)
msort l = mergeall(initlist l)
```

Figure 5.6: Merge sort in DML

converts the list to be sorted into a list of lists such that each of these lists is sorted and has length two (apart from a possible last singleton list). Function `merge2` goes through a list of lists, merging every two adjacent lists into one. Function `mergeall` iterates the application of `merge2` until a single list is obtained. The types of `initlist` and `merge2` capture the fact that the size

$$merge^c\ n_1\ n_2 = \begin{cases} n_1 = 0 \to 0 \\ n_2 = 0 \to 0 \\ n_1 > 0 \wedge n_2 > 0 \to 1 + \begin{cases} merge^c\ (n_1 - 1)\ n_2 \\ merge^c\ n_1\ (n_2 - 1) \end{cases} \end{cases}$$

$$initlist^c\ n = \begin{cases} n = 0 \to 0 \\ n = 1 \to 0 \\ n > 1 \to 1 + initlist^c\ (n - 2) \end{cases}$$

$$merge2^c\ m\ n = \begin{cases} m = 0 \wedge n = 0 \to 0 \\ m = 1 \to 0 \\ m > 1 \to merge^c\ n_1\ n_2 + merge2^c\ (m - 2)\ n_3 \\ \quad \text{where} \quad n_1 + n_2 + n_3 = n \end{cases}$$

$$mergeall^c\ m\ n = \begin{cases} m = 0 \wedge n = 0 \to 0 \\ m = 1 \to 0 \\ m > 1 \to merge2^c\ m\ n + mergeall^c(\lceil m/2 \rceil)\ n \end{cases}$$

$$msort^c\ n = initlist^c\ n + mergeall^c\ (\lceil n/2 \rceil)\ n$$

a: Extracted cost recurrences

$$merge^c\ n_1\ n_2 = n_1 + n_2$$

$$initlist^c\ n = \lfloor n/2 \rfloor$$

$$merge2^c\ m\ n = n$$

$$mergeall^c\ m\ n = \begin{cases} m \leq 1 \to 0 \\ m > 1 \to n + mergeall^c\ (\lceil m/2 \rceil)\ n \end{cases}$$

$$msort^c\ n = \lfloor n/2 \rfloor + mergeall^c\ (\lceil n/2 \rceil)\ n$$

b: Approximated cost recurrences

Figure 5.7: Cost recurrences for merge sort

measure that steers the recursion is continually halved—the index expression $\lceil n/2 \rceil$ can be encoded as $div(n, 2) + mod(n, 2)$ in the integer constraint domain we are working with.

We extract cost recurrences (Figure 5.7a), this time counting the number

of comparisons (counting calls to primitive functions works the same as counting calls to user-defined functions). The recurrences for $merge^c$, $initlist^c$ and $merge2^c(m, n)$ are easy to approximate, yielding a simplified set of recurrences displayed in Figure 5.7b. Notice how the information about halving the length of the list of lists captured in the type of `merge2` appears in $mergeall^c\ m\ n$. The solution of this recurrence equation is well-known to be $\mathcal{O}(n\ log\ m)$, which gives an overall complexity for `msort` of $\mathcal{O}(n\ log\ n)$.

An extraction scheme without access to such high-level information as provided by DML types might still provide enough *implicit* information in the extracted cost bound to derive the same bound, but the reasoning over the cost recurrence would be more involved. Basically, information about argument sizes that is encoded in the DML type and carried over into the cost recurrence with our method, would first have to be (re)proven for the cost bound.

5.4 Formal development

We now formally define the method for extracting cost recurrences from DML programs. The development is based on the theoretical view of DML presented in Section 5.2.2 and therefore only treats a monomorphic version of DML without existential types. Extending the development into a polymorphic setting is straightforward and has been omitted for the sake of conciseness. For simplicity, the formalization also does not treat mutual recursion. Extracting a cost bound from mutually recursive programs works exactly the same, but checking whether an extracted bound is indeed a recurrence becomes somewhat more tricky. For the latter, techniques such as presented in Xi's latest work [14] could be used (see Section 5.5).

A first assessment shows that our method can also be extended to existential types in a straightforward way. Essentially, the application of a function that returns a value of existential type introduces a new index variable that can be treated in the same way as new index variables introduced by pattern matching.

We start by defining the first-order fragment of DML treated by our method. For this fragment, we introduce a cost measure using a monadic translation with a cost monad. After defining a language of recurrence equations, we present the extraction algorithm. We prove its correctness by showing that extraction, if successful, indeed yields an upper bound with respect to the cost model defined by the monadic translation.

5.4.1 A first-order fragment of DML

As pointed out in Section 5.3.1, the first step of extracting cost recurrences from a first-order $\mathrm{DML}_0(\mathbb{Z})$ program is type-elaboration, which results in a $\mathrm{DML}_0^{\Pi}(\mathbb{Z})$ program of the form given in Figure 5.8 (we abbreviate a row of abstractions over $a_0 : \gamma_0, \ldots a_l : \gamma_l$ with $\lambda\vec{a} : \vec{\gamma}$, and $\langle x_0, \langle x_1, \ldots, \langle x_{i-1}, x_i\rangle\rangle\rangle$ with $\langle x_0, \ldots, x_i\rangle$). The extraction algorithm to be presented in Section 5.4.4 therefore operates on the language defined in Figure 5.8 (cf. the full language in Figure 5.4).

$$
\begin{array}{l}
\textbf{let } F_1 = \textbf{fix } F_1 : \Pi\, a_0 : \gamma_0 \ldots \Pi\, a_{l_1} : \gamma_{l_1} \,.\, \rho_{11} \to \rho_{12}. \\
\qquad\qquad \lambda \vec{a} : \vec{\gamma} \,.\, \textbf{lam } x : \rho_{11} \,.\, \mathit{body} \\
\vdots \\
\quad F_k = \textbf{fix } F_k : \Pi\, a_0 : \gamma_0 \ldots \Pi\, a_{l_k} : \gamma_{l_k} \,.\, \rho_{k1} \to \rho_{k2}. \\
\qquad\qquad \lambda \vec{a} : \vec{\gamma} \,.\, \textbf{lam } x : \rho_{k1} \,.\, \mathit{body} \\
\textbf{in } e \\
\textbf{end}
\end{array}
$$

a: Def. of functions $F_1 \ldots F_k$ in $\mathrm{DML}_0^{\Pi}(\mathbb{Z})$

$$
\begin{array}{rcl}
\mathit{body} & ::= & \textbf{case } x \textbf{ of } \langle x_0, \ldots, x_{i_k} \rangle \Rightarrow e \\
e & ::= & x \mid \langle\rangle \mid \langle e_1, e_2 \rangle \mid c[i_1] \ldots [i_n] \mid c[i_1] \ldots [i_n](e) \\
 & & \mid (\textbf{case } e \textbf{ of } ms) \mid \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} \\
 & & \mid F_i[i_1] \ldots [i_n](e) \\
p & ::= & x \mid c[a_1] \ldots [a_n] \mid c[a_1] \ldots [a_n](p) \mid \langle\rangle \mid \langle p_1, p_2 \rangle \\
ms & ::= & p \Rightarrow e \mid p \Rightarrow e \mid ms
\end{array}
$$

b: Grammar of function bodies

Figure 5.8: A first-order fragment of $\mathrm{DML}_0^{\Pi}(\mathbb{Z})$

The original semantics of DML (see Appendix 5.A.2) is defined as a natural semantics: $e \longrightarrow v$ means that e evaluates under environment Θ to value v. The semantics has a rule ev-fix for unfolding fixed-point definitions to handle recursion. For the first-order fragment of DML with its restricted form of function definition and function application used here, it is convenient to define a semantics that handles recursion using an environment of function definitions. We define a modified semantics: $e \longrightarrow_{\Theta} v$ means that e evaluates under environment Θ to value v, where Θ is a substitution mapping function names to their definitions. We show the following theorem:

Theorem 5.1 *Let p be a program of the form given in Figure 5.8. Then $p \longrightarrow v$ in the standard semantics iff $p \longrightarrow_{[]} v$ in the modified semantics.*

The definition of the modified semantics and the proof of Theorem 5.1 are deferred to Appendix 5.B.1.

5.4.2 Measuring cost of computation

One way of introducing a cost measure into functional programs is the monadic translation [6] with a cost monad. It is well-known that state can be added to a program by (1) performing a monadic translation with the state monad [10] and (2) taking the term model of the result, i.e., expanding the monadic constructs inserted by the translation to code. Similarly, using the cost monad instead

of the state monad, we can transform a program such that a cost counter is maintained.

The cost monad pairs computations that result in a value of type τ with a second component of a type $\mathbf{C}$ that expresses the cost of the computation; all we need to know is that $\mathbf{C}$ is an ordered Abelian monoid[6] $(\mathbf{C}, +, \mathbf{0}, \leq)$. We write $\mathsf{C}\,\alpha$ as a type abbreviation for $\alpha \times \mathbf{C}$. A call-by-value monadic translation with a cost monad that is based on C turns a function of type $\Pi\, a_0 : \gamma_0 \ldots \Pi\, a_k : \gamma_k \,.\, \rho_1 \to \rho_2$ into a function of type $\Pi\, a_0 : \gamma_0 \ldots \Pi\, a_k : \gamma_k \,.\, \rho_1 \to \mathsf{C}\,\rho_2$. The intended meaning is that the transformed function not only returns the result value, but also the cost of computing it.

The cost monad can be defined by specifying two language constructs, $\mathbf{val}^{\mathsf{C}}$ and $\mathbf{let}^{\mathsf{C}}$, which a monadic translation inserts into a program text. The typing rule for $\mathbf{val}^{\mathsf{C}}$ is

$$\frac{\phi; \Gamma \vdash e : \rho}{\phi; \Gamma \vdash \mathbf{val}^{\mathsf{C}}\ e : \mathsf{C}\,\rho}\ \textsf{(ty-monadic-val)}$$

The construct $\mathbf{val}^{\mathsf{C}}$ is used to inject a value $v : \rho$ into $\mathsf{C}\,\rho$ as $\langle v, \mathbf{0}\rangle$—values do not require any computation and thus incur no cost. The typing rule for $\mathbf{let}^{\mathsf{C}}$ is

$$\frac{\phi; \Gamma \vdash e_1 : \mathsf{C}\,\rho_1 \qquad \phi; \Gamma, x : \rho_1 \vdash e_2 : \mathsf{C}\,\rho_2}{\phi; \Gamma \vdash \mathbf{let}^{\mathsf{C}}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : \mathsf{C}\,\rho_2}\ \textsf{(ty-monadic-let)}$$

In $(\mathbf{let}^{\mathsf{C}}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end})$, the expression e_1 is evaluated to a result v_1 wrapped with a cost z_1. To calculate e_2, the unwrapped v_1 is used, yielding $\langle v_2, z_2\rangle$. The final result of the let expression is $\langle v_2, z_1 + z_2\rangle$.

The monadic translation provides only the infrastructure for tracking cost, but does not assign costs to any program constructs. This assignment of costs is done by inserting a monadic construct $\mathbf{cost}_z$ with typing rule

$$\frac{\phi; \Gamma \vdash e : \mathsf{C}\,\rho \qquad z \in \mathbf{C}}{\phi; \Gamma \vdash \mathbf{cost}_z\ e : \mathsf{C}\,\rho}\ \textsf{(ty-monadic-cost)}$$

into the transformed program. $\mathbf{cost}_z$ is particular to the cost monad: It adds z to the cost component of the value it is applied to.

A *cost-conscious* version of a program, i.e., a program that keeps track of the cost incurred by calls to user-defined functions and uses of constructors, is generated as follows: We first perform a monadic translation of the program and then enclose (1) each application of a user-defined function F with $\mathbf{cost}_{\mathbf{c}_F}$, and (2) each application of a constructor c with $\mathbf{cost}_{\mathbf{c}_c}$.

Figure 5.9 displays the combined translation $(\cdot)^*$ for function bodies. A program p of the form given in Figure 5.8 is translated into p^* by applying the monadic translation to all function bodies and the body of the program, and changing every result type ρ_{l2} in the type annotation of the fixed-point definition to $\mathsf{C}\,\rho_{l2}$. The cost-conscious version can easily be expressed in $\mathrm{DML}_0^{\Pi}(C)$: Figure 5.10 shows how to expand $\mathbf{val}^{\mathsf{C}}$, $\mathbf{let}^{\mathsf{C}}$ and $\mathbf{cost}_z$.

[6] In an ordered monoid, the ordering $\leq$ is compatible with the monoid's operation, i.e., if $a \leq b$ then $a + c \leq b + c$. Relevant examples of ordered monoids are $(\mathbb{N}, +, 0, \leq)$ and $(\mathbb{R}, +, 0, \leq)$.

The following theorem shows that the cost translation is well-behaved:

Theorem 5.2 *Let* $\cdot;\cdot \vdash \mathsf{p} : \rho$ *be derivable. Then*

1. *judgment* $\cdot;\cdot \vdash \mathsf{p}^* : \mathsf{C}\,\rho$ *is derivable.*
2. $\mathsf{p} \longrightarrow_{[]} v$ *iff* $\mathsf{p}^* \longrightarrow_{[]} \langle v, z\rangle$ *for some* $z \in \mathbf{C}$.

The proof is deferred to Appendix 5.B.2.

$$
\begin{array}{lcl}
x^* & = & \mathbf{val}^{\mathsf{C}}\ x \\
\langle\rangle^* & = & \mathbf{val}^{\mathsf{C}}\ \langle\rangle \\
\langle e_1, e_2\rangle^* & = & \mathbf{let}^{\mathsf{C}}\ x_1 = {e_1}^*\ \mathbf{in} \\
 & & \mathbf{let}^{\mathsf{C}}\ x_2 = {e_2}^* \\
 & & \mathbf{in}\ \mathbf{val}^{\mathsf{C}}\ \langle x_1, x_2\rangle\ \mathbf{end} \\
(c[i_1]\ldots[i_k])^* & = & \mathbf{cost}_{\mathbf{c}_c}\ (\mathbf{val}^{\mathsf{C}}\ (c[i_1]\ldots[i_k])) \\
(c[i_1]\ldots[i_k](e))^* & = & \mathbf{let}^{\mathsf{C}}\ x = e^* \\
 & & \mathbf{in}\ \mathbf{cost}_{\mathbf{c}_c}\ (\mathbf{val}^{\mathsf{C}}\ (c[i_1]\ldots[i_k](x)))\ \mathbf{end} \\
(\mathbf{case}\ e\ \mathbf{of}\ ms)^* & = & \mathbf{let}^{\mathsf{C}}\ x = e^* \\
 & & \mathbf{in\ case}\ x\ \mathbf{of}\ ms^*\ \mathbf{end} \\
(p \Rightarrow e \mid ms)^* & = & p \Rightarrow e^* \mid ms^* \\
(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end})^* & = & \mathbf{let}^{\mathsf{C}}\ x = {e_1}^* \\
 & & \mathbf{in}\ {e_2}^*\ \mathbf{end} \\
(F[i_1]\ldots[i_k](e))^* & = & \mathbf{let}^{\mathsf{C}}\ x = e^* \\
 & & \mathbf{in}\ \mathbf{cost}_{\mathbf{c}_c}\ (F[i_1]\ldots[i_k](x))\ \mathbf{end}
\end{array}
$$

Figure 5.9: Monadic translation of function bodies

5.4.3 A language of recurrence equations

The language of recurrence equations is based on the natural numbers part $\mathbb{N}$ of the constraint domain $\mathbb{Z}$ and the cost domain $\mathbf{C}$. Natural numbers and tuples thereof serve as abstractions of input size, and therefore are used as arguments of recurrence equations. The result of a recurrence equation represents cost of computation and is expressed in $\mathbf{C}$.

Syntax and types

We describe a system of recurrence equations with the language given in Figure 5.11. Because we extract recurrences from programs that are not mutual

$$
\begin{array}{lcl}
\mathbf{val}^{\mathsf{C}}\ e & \equiv & \langle e, \mathbf{0}\rangle \\
\mathbf{let}^{\mathsf{C}}\ x = e_1 & \equiv & \mathbf{case}\ e_1\ \mathbf{of} \\
\mathbf{in}\ e_2 & & \quad \langle x, z_1\rangle \Rightarrow\ \mathbf{case}\ e_2\ \mathbf{of} \\
\mathbf{end} & & \quad\quad \langle v, z_2\rangle \Rightarrow \langle v, z_1 + z_2\rangle \\
\mathbf{cost}_z\ e & \equiv & \mathbf{case}\ e\ \mathbf{of} \\
 & & \quad \langle x, z'\rangle \Rightarrow\ \langle x, z + z'\rangle
\end{array}
$$

Figure 5.10: Monadic constructs as syntactic sugar

recursive, neither is a system of recurrence equations, i.e., a body t_l may only contain recurrence-equation names $F_1^c \ldots F_l^c$. Conditionals, which so far have been pretty printed, are introduced with the keyword **cond** followed by a number of branches. Within a branch, the first constraint Φ_1 represents the guard of the branch, whereas the second constraint Φ_2 represents a *where*-clause. The scope of the quantification (we write $\vec{a} : \vec{\sigma}$ as shorthand for the quantification over variables $a_1 : \sigma_1\ \ldots a_k : \sigma_k$) extends both over Φ_2 and the branch body t. For $\forall \vec{a} : \vec{\sigma}.\Phi_2$ we require that for any interpretation of its free variables, there are only finitely many instantiations of $\vec{a}$ such that Φ_2 is satisfied; this requirement is met for recurrence equations extracted from DML programs in which the data types are enriched "sensibly" as required in the beginning of Section 5.3.1.

$$
\begin{array}{llcl}
\text{index types} & \sigma & ::= & \mathbb{N} \mid \mathbf{1} \mid \sigma_1 \times \sigma_2 \\
\text{types} & \nu & ::= & \mathbf{C} \mid \sigma \to \nu \\
\text{definitions} & \mathfrak{E} & ::= & F_1^c\ a_0\ a_1 \ldots a_{l_1} = t_1 \\
 & & & \quad\vdots \\
 & & & F_k^c\ a_0\ a_1 \ldots a_{l_k} = t_k \\
\text{body} & t & ::= & z \mid t_1 + t_2 \mid F^c\ \vec{\imath} \mid (\mathbf{cond}\ brs) \\
\text{index objects} & i, j & ::= & a \mid f(i) \mid \langle\rangle \mid \langle i, j\rangle \mid \mathbf{fst}(i) \mid \mathbf{snd}(i) \\
\text{branches} & brs & ::= & br \mid br \mid brs \\
\text{branch} & br & ::= & (\Phi_1 \to \forall \vec{a} : \vec{\sigma}.\Phi_2 \to t)
\end{array}
$$

Figure 5.11: Syntax of recurrence equations

The rationale behind the shape of recurrence equation types is that for a function F_l of type

$$\Pi\, a_0 : \gamma_0\,.\,\Pi\, a_1 : \gamma_1 \ldots \Pi\, a_k : \gamma_k\,.\,\rho_1 \to \rho_2,$$

the associated cost recurrence F_l^c should have type

$$\widetilde{\gamma_0} \to \widetilde{\gamma_1} \to \ldots \widetilde{\gamma_k} \to \mathbf{C},$$

where $\widetilde{\cdot}$ maps an index sort to the associated index type. For example $\{a : \mathbb{N} \times \mathbb{N} \mid \mathbf{fst}(a) \leq \mathbf{snd}(a)\}$ is mapped to $\mathbb{N} \times \mathbb{N}$.

The formal definition of $\widetilde{\cdot}$ and its extension to contexts Γ is deferred to Appendix 5.B.3.

The body of a recurrence equation F_k^c is typed using the judgment

$$\varphi; \Delta \vdash e : \nu$$

in which φ maps index types σ to index variables, and Δ assigns recurrence-equation types to function names. Figure 5.12 gives typing rules, all of which are straightforward. So are the rules for checking the type of an index object ($\varphi \vdash i : \sigma$) and the wellformedness of a constraint ($\varphi \vdash \Phi$), which have been omitted.

Based on the typing rules for the body of a recurrence equation, we define what it means for a system of recurrence equations to be well-typed with respect to the program they have been extracted from.

Definition 5.3 *Let* p *be a program and context* Γ *assign every* F_l *defined in* p *to its declared type. A system of recurrence equations* $\mathfrak{E}$ *is well-typed with respect to* p *if for every*

$$F : \Pi\, a_0 : \gamma_0 \,.\, \Pi\, a_1 : \gamma_1 \ldots \Pi\, a_k : \gamma_k \,.\, \rho_1 \to \rho_2$$

defined in p *, for the extracted recurrence equation*

$$F^c\ a_0 \ldots\ a_k\ =\ t$$

we have

$$a_0 : \widetilde{\gamma_0}, \ldots, a_k : \widetilde{\gamma_k}; \widetilde{\Gamma} \vdash t : \mathbf{C}.$$

Semantics

We give a simple denotational semantics to the language of recurrence equations. A recurrence equation defining a function of type

$$\sigma_0 \to \sigma_1 \to \ldots \sigma_k \to \mathbf{C}$$

is interpreted in the function domain

$$[\mathcal{I}[\![\sigma_0]\!] \to \mathcal{I}[\![\sigma_1]\!] \to \ldots \mathcal{I}[\![\sigma_k]\!] \to \mathbf{C}_\perp].$$

(Because all $\mathcal{I}[\![\sigma_l]\!]$ are discrete cpos, such functions are necessarily continuous).

Here $\mathcal{I}[\![\cdot]\!]$ is the canonical semantics given to ground index objects i and index types σ by the constraint domain $\mathbb{Z}$; for an index substitution θ that maps index variables to ground index objects, we write $\mathcal{I}[\![\theta]\!]$ for the corresponding mapping into $\mathbb{Z}$. With $\mathbf{C}_\perp$ we denote the domain that results from interpreting $\mathbf{C}$ as a discrete domain and lifting it in the canonical way—in the semantics definition

$$\frac{}{\varphi; \Delta \vdash z : \mathbf{C}} \text{ (ty-re-const)}$$

$$\frac{\varphi; \Delta \vdash e_1 : \mathbf{C} \qquad \varphi; \Delta \vdash e_2 : \mathbf{C}}{\varphi; \Delta \vdash e_1 + e_2 : \mathbf{C}} \text{ (ty-re-plus)}$$

$$\frac{\begin{array}{c}\Delta(F) = \sigma_0 \to \ldots \to \sigma_k \to \mathbf{C} \\ \varphi \vdash i_0 : \sigma_0 \quad \ldots \quad \varphi \vdash i_k : \sigma_k\end{array}}{\varphi; \Delta \vdash F^c\ i_0 \ldots i_k : \mathbf{C}} \text{ (ty-re-app)}$$

$$\frac{\begin{array}{c}\varphi \vdash \Phi_{11} \quad \varphi, \vec{a_1} : \vec{\sigma_1} \vdash \Phi_{12} \quad \varphi, \vec{a_1} : \vec{\sigma_1}; \Delta \vdash e_1 : \mathbf{C} \\ \vdots \\ \varphi \vdash \Phi_{k1} \quad \varphi, \vec{a_k} : \vec{\sigma_k} \vdash \Phi_{k2} \quad \varphi, \vec{a_k} : \vec{\sigma_k}; \Delta \vdash e_k : \mathbf{C}\end{array}}{\begin{array}{l}\varphi; \Delta \vdash (\mathbf{cond}\ \Phi_{11} \to \forall \vec{a_1} : \vec{\sigma_1}.\Phi_{12} \to e_1 \\ \qquad \vdots \\ \qquad \mid \Phi_{k1} \to \forall \vec{a_k} : \vec{\sigma_k}.\Phi_{k2} \to e_k) : \mathbf{C}\end{array}} \text{ (ty-re-cond)}$$

Figure 5.12: Typing rules for recurrence equations

$$\begin{array}{rcl}
\mathcal{T}[\![z]\!]\Psi\theta & = & \llcorner z \lrcorner \\
\mathcal{T}[\![t_1 + t_2]\!]\Psi\theta & = & \mathcal{T}[\![t_1]\!]\Psi\theta +_\perp \mathcal{T}[\![t_2]\!]\Psi\theta \\
\mathcal{T}[\![F_l^c\ i_0 \ldots i_k]\!]\Psi\theta & = & \Psi(F_l^c)\ (\mathcal{I}[\![i_0[\theta]]\!]) \ldots (\mathcal{I}[\![i_k[\theta]]\!]) \\
\mathcal{T}[\![\mathbf{cond}\ br_0 \mid \ldots \mid br_k]\!]\Psi\theta & = & max_\perp\{\mathcal{B}[\![br_0]\!]\Psi\theta, \ldots, \mathcal{B}[\![br_k]\!]\Psi\theta\} \\
\mathcal{B}[\![(\Phi_1 \to \forall \vec{a} : \vec{\sigma}.\Phi_2 \to t)]\!]\Psi\theta & = & \begin{cases} \llcorner \mathbf{0} \lrcorner & \text{if } \Phi_1[\theta] \text{ does not hold} \\ max_\perp\{\mathcal{T}[\![t]\!]\Psi\theta[\vec{a} \mapsto \vec{z}] \mid & \\ \qquad \vec{z} \in \mathcal{I}[\![\vec{\sigma}]\!]\ \wedge \Phi_2[\theta[\vec{a} \mapsto \vec{z}]]\} & \\ \qquad \text{if } \Phi_1[\theta] \text{ holds.} & \end{cases}
\end{array}$$

Figure 5.13: Semantics of recurrence equations

we also mark operations on $\mathbf{C}$ that have been lifted in the canonical way by subscripting them with $\perp$.

Figure 5.13 gives a semantics $\mathcal{T}[\![\cdot]\!]\Psi\theta$ for recurrence-equation expressions, where Ψ is a mapping from recurrence-equation names to functions and θ an index substitution that maps index variables to ground index objects. The semantics treats a conditional by taking the maximum of the values returned by the branches of the conditional. If the guard of a branch does not hold,

the branch returns **0**. Otherwise, all possible values for the universally quantified index variables in the branch are tried out and the maximum is returned. Because, as required in Section 5.4.3, there is at most a finite number of such values, the semantics of a branch is well-defined (we assume that $max_{\perp}\emptyset = \llcorner \mathbf{0} \lrcorner$).

Given the definition of a recurrence equation

$$F^c a_0\ a_1 \ldots a_k = t$$

and a mapping Ψ that ranges over all names of recurrence-equations declared previously to F^c (recall that we do not consider mutually recursive systems of recurrence equations), then the semantics of the defined function is the fixed point of the functional

$$\lambda\mathcal{F}.\lambda n_0\ n_1 \ldots n_k.\mathcal{T}[\![t]\!](\Psi[F^c \mapsto \mathcal{F}])[a_0, \ldots, a_k \mapsto n_0, \ldots n_k].$$

This semantics of a single recurrence equation extends naturally to a system $\mathfrak{E}$ of recurrence equations and yields a mapping from recurrence-equation names to functions; we write $\mathcal{S}[\![\mathfrak{E}]\!]$.

5.4.4 The extraction algorithm

We extract cost recurrences from $\mathrm{DML}_0^{\Pi}(\mathbb{Z})$ type derivations. A $\mathrm{DML}_0^{\Pi}(\mathbb{Z})$ typing judgment is of the form

$$\phi; \Gamma \vdash e : \tau,$$

where ϕ is an index context and Γ is a variable context (see Appendix 5.A.1 for details). The central part of the extraction algorithm operates on the type derivations for the bodies of function definitions $F_1 \ldots F_n$: From an expression e of first-order type that occurs within the body of a function definition, a recurrence-equation expression t of type $\mathbf{C}$ is extracted. The algorithm is defined in form of a judgment

$$\phi; \Gamma \vdash e : \rho \blacktriangleright t.$$

Consider the following definition of a function F (see Figure 5.8):

$$\begin{array}{l}\textbf{let}\ F = \textbf{fix}\ F : \Pi\, a_0 : \gamma_0 \ldots \Pi\, a_l : \gamma_l\, .\, \rho_1 \to \rho_2. \\ \qquad\qquad \lambda \vec{a} : \vec{\gamma}\, .\, \textbf{lam}\ x : \rho_1\, .\, \textbf{case}\ x\ \textbf{of}\ \langle x_0, \ldots, x_k \rangle \Rightarrow e\end{array}$$

Assuming that e is typed as $\phi; \Gamma \vdash e : \rho_2$, the extracted recurrence equation is $F^c\ a_0 \ldots a_l = t$, where t results from $\phi; \Gamma \vdash e : \rho_2 \blacktriangleright t$.

Before we give a complete description of the extraction of cost recurrences, we examine how an index context can be turned into a constraint over the declared index variables.

Turning an index context into a constraint

$\mathrm{DML}_0^{\Pi}(\mathbb{Z})$ expressions are typed under a "normal" context and an index context of form

$$\phi ::= \cdot \mid \phi, a : \gamma \mid \phi, P$$

(See Figure 5.3). Basically, the index context collects constraints over index variables. It is straightforward to define a function $\mathcal{C}$ that rewrites an index context ϕ into a constraint Φ such that sort definitions are "flattened out", i.e.,

$$\mathcal{C}(a : \{k : \mathbb{N} \mid k > 0\}, b : \{k : \{k' : \mathbb{N} \mid k' > 1\} \mid k \geq a\}) = a > 0 \wedge b > 1 \wedge b \geq a$$

A precise definition of $\mathcal{C}$ is deferred to Appendix 5.B.3.

A useful operation over constraints is to *project out* a certain set of variables, i.e., "hide" all remaining variables by existential quantification. We write $\exists \vec{a}.\Phi$ for the constraint that results from existentially quantifying over the variables $\vec{a}$ in Φ. With a constraint solver such as used for DML type-checking, existentially quantified variables usually are simplified away.

Often equalities can be derived from a constraint Φ. Let $\vec{a}$ be a subset of the free variables in Φ; we define a substitution $\theta := mk\_subst_{\vec{a}}(\Phi)$ as follows: For all $a \in \vec{a}$ such that i is an index expression without free variables in $\vec{a}$, and $a = i$ can be derived from Φ, we have $\theta(a) = i$.

Definition of the extraction algorithm

The definition of the judgment $\phi; \Gamma \vdash e : \rho \blacktriangleright t$ is given in Figure 5.14.

Most rules are fairly straightforward:

- Accessing a variable or the unit value has no cost.
- For pairs and let expressions, the total cost is the sum of the costs incurred by their subexpressions.
- The cost of executing a user-defined function F has three components: the cost of evaluating its argument, the cost $\mathbf{c}_F$ of calling the function, and the cost incurred by evaluating the function. For the latter component, a recursive call with the appropriate arguments is generated.
- The use of a constructor c costs $\mathbf{c}_c$ plus the cost of evaluating a possible argument.

A case expression is handled by converting it into a conditional. The real heart of the algorithm is the rule that handles branches of case expressions. This rule extracts a conditional branch from a type derivation for

$$\phi; \Gamma \vdash (p \Rightarrow e) : \rho' \Rightarrow \rho.$$

This judgment types the branch of a case expression that matches a pattern p against a value of type ρ'; the expression e in the branch has the same type ρ as the total case expression. We go through the premises of the corresponding extraction rule one by one:

1. The judgment $p \downarrow \rho' \rhd (\phi'; \Gamma')$ (see Figure 5.16 on page 167) is defined as part of the $\mathrm{DML}_0^{\Pi}(C)$ type-checking rules. It generates an index context

$$\frac{\phi;\Gamma \vdash e : \tau_1 \blacktriangleright t \quad \phi \models \tau_1 \equiv \tau_2}{\phi;\Gamma \vdash e : \tau_2 \blacktriangleright t} \text{ (extr-eq)}$$

$$\frac{}{\phi;\Gamma \vdash x : \tau \blacktriangleright \mathbf{0}} \text{ (extr-var)} \qquad \frac{}{\phi;\Gamma \vdash \langle\rangle : \mathbf{1} \blacktriangleright \mathbf{0}} \text{ (extr-unit)}$$

$$\frac{\phi;\Gamma \vdash e_1 : \tau_1 \blacktriangleright t_1 \quad \phi;\Gamma \vdash e_2 : \tau_2 \blacktriangleright t_2}{\phi;\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2 \blacktriangleright t_1 + t_2} \text{ (extr-pair)}$$

$$\frac{\phi;\Gamma \vdash e_1 : \rho_1 \blacktriangleright t_1 \quad \phi;\Gamma, x : \rho_1 \vdash e_2 : \rho_2 \blacktriangleright t_2}{\phi;\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : \rho_2 \blacktriangleright t_1 + t_2} \text{ (extr-let)}$$

$$\frac{\phi;\Gamma \vdash e : \rho_1 \blacktriangleright t}{\phi;\Gamma \vdash F[i_1]\ldots[i_k](e) : \rho_2 \blacktriangleright t + \mathbf{c}_F + (F^c i_1 \ldots i_k)} \text{ (extr-app)}$$

$$\frac{}{\phi;\Gamma \vdash c[i_1]\ldots[i_n] : \rho \blacktriangleright \mathbf{c}_c} \text{ (ty-cons-wo)}$$

$$\frac{\phi;\Gamma \vdash e : \tau \blacktriangleright t}{\phi;\Gamma \vdash c[i_1]\ldots[i_n](e) : \rho \blacktriangleright t + \mathbf{c}_c} \text{ (ty-cons-w)}$$

$$\frac{\begin{array}{c}\phi;\Gamma \vdash e : \rho' \blacktriangleright t \\ \phi;\Gamma \vdash (p_0 \Rightarrow e_0) : \rho' \Rightarrow \rho \blacktriangleright br_1 \\ \vdots \\ \phi;\Gamma \vdash (p_k \Rightarrow e_k) : \rho' \Rightarrow \rho \blacktriangleright br_k\end{array}}{\begin{array}{l}\phi;\Gamma \vdash (\mathbf{case}\ e\ \mathbf{of}\ p_0 \Rightarrow e_0 \mid \ldots \mid p_k \Rightarrow e_k) : \rho \\ \quad \blacktriangleright t + (\mathbf{cond}\ br_1 \mid br_2 \mid \ldots \mid br_k)\end{array}} \text{ (ty-case)}$$

$$\frac{\begin{array}{l}p \downarrow \rho' \rhd (\phi';\Gamma') \\ \phi,\phi';\Gamma,\Gamma' \vdash e : \rho \blacktriangleright t \\ \Phi_1 = \exists(\mathbf{dom}(\phi,\phi')\backslash\mathbf{var}(\rho')).\mathcal{C}(\phi,\phi') \\ \theta = mk\_subst_{\mathbf{dom}(\phi')}(\mathcal{C}(\phi')) \\ \Phi_2 = \exists(\mathbf{dom}(\theta)).\mathcal{C}(\phi')\end{array}}{\begin{array}{l}\phi;\Gamma \vdash (p \Rightarrow e) : \rho' \Rightarrow \rho \\ \quad \blacktriangleright \Phi_1 \rightarrow \forall(\mathbf{dom}(\phi')\backslash\mathbf{dom}(\theta)).\Phi_2 \rightarrow t[\theta]\end{array}} \text{ (ty-branch)}$$

Figure 5.14: Extraction algorithm

ϕ' and a variable context Γ' that describe the index variables and normal variables occurring in the pattern p.

For example, after translation into $\mathrm{DML}_0^{\Pi}(\mathbb{Z})$, the pattern of the second branch in `flatten` (Section 5.3.2) is $\mathit{lcons}[m'][n_1][n_2]\langle xs, \mathit{rest}\rangle$; type-checking under index context $\phi = m : \mathbb{N}, n : \mathbb{N}$ generates the following index context:

$$\phi' = m' : \mathbb{N}, n_1 : \mathbb{N}, n_2 : \mathbb{N}, m = m' + 1, n = n_1 + n_2.$$

2. $\mathrm{DML}_0^{\Pi}(C)$ type-checking types the branch expression under the contexts ϕ, ϕ' and Γ, Γ'. From the resulting type derivation, a recurrence-equation expression t is extracted.

   For the second branch of `flatten`, this yields

   $$t = 2 + (\mathit{append}^c\ n_1\ n_2) + (\mathit{flatten}^c\ m'\ n_2)$$

3. The guard of the branch Φ_1 is derived by projecting out from $\mathcal{C}(\phi, \phi')$ over the index variables contained in ρ'.

   In `flatten`, the only index variable in $\rho' = \mathsf{llist}(m,n)$ are m and n; projecting them out yields $m > 0$. For n there is no information, because we only know that $n \in \mathbb{N}$, and (as pointed out in Section 5.3.1) we require all indices to be of subsorts of $\mathbb{N}$ anyways.

4. All index variables declared in ϕ' that can be expressed in terms of index variables from ϕ should be are eliminated using substitution θ: In the branch returned by the rule, the body is $t[\theta]$ rather than t.

   For the second branch of `flatten`, we can infer that $m' = m - 1$, so $\theta = [m' \mapsto m - 1]$.

5. Index variables declared in ϕ' for which no equality constraint can be derived have to be restricted. This is done by (1) hiding the variables $\mathbf{dom}(\theta)$ via existential quantification in $\mathcal{C}(\phi')$, and (2) universally quantifying over the remaining variables of $\mathbf{dom}(\phi')$, binding variables both in Φ_2 and in $t[\theta]$. Conjuncts in Φ_2 containing none of the universally quantified variables can be dropped (such conjuncts are guaranteed to be satisfied whenever Φ_1 is satisfied).

   For the second branch of `flatten`, existential quantification over m' in ϕ' yields, after normalization, $m > 0 \wedge n = n_1 + n_2$. We universally quantify over n_1 and n_2, and drop the conjunct $m > 0$.

To complete the running example: The second branch of `flatten` gives rise to the following branch of the conditional in $\mathit{flatten}^c$:

$$\begin{aligned} m > 0 \rightarrow \forall n_1, n_2 : \mathbb{N}. n_1 + n_2 = n \rightarrow \\ 2 + (\mathit{append}^c\ n_1\ n_2) + (\mathit{flatten}^c\ (m-1)\ n_2) \end{aligned}$$

5.4.5 Checking whether the bound is a recurrence

It remains to check whether we really extract a recurrence, i.e., a function defined in terms of its value on smaller arguments. For the presented language without mutual recursion (for handling mutual recursion, Xi's most recent work [14] could be adapted—see Section 5.5), this can be conveniently done during extraction when generating a recursive call $F^c\ i_1 \dots i_k$ inside the body that defines $F^c\ a_1 \dots a_k$: The extraction yields a recurrence if for every such call,

$$\langle i_1, \dots, i_n \rangle < \langle a_1, \dots, a_n \rangle$$

can be derived from the collected constraints for a well-founded order $<$ on tuples. In practice, one could for example fix the usual lexicographic ordering, requiring the user to enrich data correspondingly, or leave the user a choice as to which ordering should be used.

5.4.6 Correctness

Theorem 5.4 *Let functions $F_1, \dots, F_k$ be a well-typed block of function definitions. Let $\mathfrak{E}$ be the system of recurrence equations for $F_1^c, \dots, F_k^c$ extracted from these definitions. Let p be a well-typed program consisting of these function definitions $F_1, \dots, F_k$ and a program body $F_l[z_1] \dots [z_{l'}](u)$, where $z_1, \dots, z_{l'}$ are ground index objects and u is a value. Then:*

1. *$\mathfrak{E}$ is well-typed with respect to p.*
2. *If $\mathcal{T}[\![F_l^c\ z_1 \dots z_{l'}]\!](\mathcal{S}[\![\mathfrak{E}]\!])[] = \llcorner z' \lrcorner$ for some $z' \in \mathbf{C}$, then there exists a value v and a $z \in \mathbf{C}$ with $z \leq z'$ such that p^* evaluates to $\langle v, z \rangle$.*
3. *If the test described in Section 5.4.5 has been passed during the extraction of $\mathfrak{E}$, then the denotation of $F_l^c z_1 \dots z_k$ under environment $\mathcal{S}[\![\mathfrak{E}]\!]$ is guaranteed to yield $\llcorner z' \lrcorner$ for some $z' \in \mathbf{C}$ rather than $\bot$.*

Part 1 and 3 of the theorem capture that the extracted system of cost bounds is well-formed: The system is well-typed, and the application of a bound F_l^c, to some legal input $z_1, \dots z_l$ (*legal* in the sense that $F_l[z_1] \dots [z_l](u)$ type-checks for some u) is well-defined. Because the translation $(\cdot)^*$ (see Section 5.4.2) captures our cost model—in the translated program, a cost counter z is calculated together with the actual result—part 2 states that this bound indeed is a cost bound for the execution of $F_l[z_1] \dots [z_l](u)$. The proof of Theorem 5.4 is deferred to Appendix 5.B.4.

5.5 Related work

We discuss related work regarding automated complexity analysis and type systems.

Le Métayer's ACE system [4] automatically extracts cost bounds for a subset of FP, expressing the extracted bounds as FP programs. The system is based

on program transformation. The first step transforms the original program into a *step-counting* version, i.e., a program that takes the same arguments, but returns the cost of computation for these arguments rather than the result. Conceptually, this transformation corresponds to a monadic translation with the cost monad as presented in Section 5.4.2, where the cost component is projected out from the final result. Subsequent transformation steps try to transform the step-counting version into a composition of a cost bound and a measure function, where the measure function is composed from selector functions and the *length* function for lists. The principal goal of the ACE system is to eliminate recursion in the cost bound, which corresponds to solving recurrences; the system's library holds more than 1000 transformation rules, many of them tailored to recognize patterns of recursion. The process of finding a measure function is interleaved with the process of eliminating recursion and cannot easily be decoupled. Pointing the system to a given measure thus seems difficult. In contrast, our method separates concerns: The user can specify an appropriate measure using dependent types, but no attempts are made to solve the extracted cost recurrence.

Sands [9] treats cost analysis for higher-order call-by-name languages: Cost bounds are extracted by program transformation and reasoning over programs; his method can be seen as an extension of Le Métayer's overall approach. Sands focuses on the complications for cost analysis caused by higher-order functions and call-by-name evaluation. No special concern is given to the abstraction of data to data size, which is the main concern of our work.

Rosendahl [8] develops a system that uses abstract interpretation and program transformation techniques to extract cost bounds from a first-order subset of Scheme. An abstract interpretation is used to extend the set of S-expressions with partially known structures—unknown parts of a structure are represented by a special token that stand for all possible structures. Size measures are expressed through "inverse size functions": For a given size, an inverse size function returns a partially known structure that approximates all data structures of that size. For example, for lists, the inverse size function generates for size n a list containing n times the special token representing all possible structures. An initial cost bound for a program is achieved by (1) composing a suitable inverse size function with the abstract interpretation of the step-counting version of the program and (2) taking the term model. Program transformation is then used to simplify this initial cost bound as much as possible. Liu and Gómez [5] propose a method based on Rosendahl's work in which they use advanced program-transformation techniques to make the cost bound more efficient and more accurate. Both methods requires the user to define the abstraction from data to data size. However, with dependent types, measures can be expressed that are impossible to define with an inverse size function—the measure used in Section 5.3.2 for analyzing `flatten` is one example.

Reistad and Gifford [7] use an effect type system for automatically inferring cost estimates of functional programs written with combinators such as map and fold. Two indexed data-types, lists and vectors, are built into the type system. The effects associated with function types are cost expressions that may

depend on indices of list/vector arguments and on cost expressions associated with function arguments. The main focus of Reistad and Gifford is to guide parallelization of programs with the inferred cost estimates.

Crary and Weirich [3] present a decidable type system for the specification and certification of resource consumption in the setting of Typed Assembly Language. The type system simulates dependent types using sum and inductive kinds. Essentially, it allows the user to annotate function arrows with resource bounds (e.g., time bounds or space bounds) in terms of the shape of data arguments and of cost bounds associated with function arguments. The focus lies on the certification of resource bounds through type checking rather than the derivation of resource bounds.

Chin and Khoo [2] propose sized types in which size information is expressed with Presburger formulas. They use a a constraint solver to infer size information. It is very likely that complexity analysis in the style of this paper could be integrated into their setting.

Recently, Xi [14] presented an extension of the DML type system that allows program termination verification. The basic observation is the same as for our work, namely that DML types can be used to encode a notion of input size. For each function, the programmer supplies a metric in terms of the input indices; if type checking succeeds, the metrics are guaranteed to specify a termination order. Both higher-order functions and general recursion are handled. We believe that our work would benefit substantially from reformulating it in this extended type system: As an immediate benefit, extracted cost bounds could be easily verified to be recurrences also for general recursion. Also an extension of this work to higher-order functions, if possible, should be easier within Xi's new type system.

5.6 Conclusion

We have presented a method for automatically extracting cost recurrences from first-order DML programs. The distinct feature of our method is the use of dependent types to describe a size measure that abstracts from data to data size. The user has to choose an appropriate size measure for our method to successfully extract a cost recurrence. Because of the high expressiveness of its types, DML offers high flexibility for tailoring size measures. The required DML type annotations usually are easy to find: Because size measures encode shape information of data types, they closely correspond to the programmer's intuitive understanding of how his program works. Our method harnesses this intuition for automatic cost analysis.

Acknowledgments I am indebted to Olivier Danvy, Julia Lawall and Zhe Yang for their encouragement and fruitful discussions on the subject of this work. Further thanks are due to Olivier Danvy, Andrzej Filinski and Julia Lawall for their numerous constructive comments. I am also grateful to the anonymous referees and Neil Jones for further comments.

5.A DML

In the following we give a short overview over the formalization of $\mathrm{DML}_0^{\Pi}(C)$ that is used in this article. We gloss over details such as type-formation rules, well-formedness of contexts, and type-equivalence. The complete formalization can be found in Xi's PhD thesis [12, Chapters 2–4].

5.A.1 DML typing rules

A typing judgment for $\mathrm{DML}_0^{\Pi}(C)$ has the form

$$\phi; \Gamma \vdash e : \tau,$$

where ϕ is an index context and Γ a (normal) context; typing is with respect to a signature $\mathcal{S}$ that assigns types to constructors. In the following, we focus on typing rules that treat indices—the remaining rules are fairly standard.

The treatment of indices is based on a judgment $\phi \vdash i : \gamma$ that expresses that $i : \gamma$ can be derived from the index context ϕ; to establish this judgment, constraint solving is required. The judgment is used, for example, to express type-soundness of an index substitution θ under a context ϕ as $\phi \vdash \theta : \phi'$ (Figure 5.15a), which basically says that, assuming the context given by ϕ substitution θ assigns to all index variables declared in ϕ' an index object of the declared sort. The following lemma [12, Chapter 3] shows a useful link between type-soundness with respect to an index context ϕ and satisfyability with respect to ϕ:

Lemma 5.5 *Let ϕ and ϕ' be index contexts and θ an index substitution such that $\phi \vdash \theta : \phi'$ is derivable. Then, if $\phi, \phi' \models \Phi$ is derivable, also $\phi \models \Phi[\theta]$ is derivable.*

Figure 5.15b defines a corresponding judgment for general substitutions; it is straightforward to show that if $\phi; \Gamma \vdash \theta : (\phi'; \Gamma')$ holds for a substitution θ, then $\phi \vdash \theta_\phi : \phi'$ holds for its restriction θ_ϕ to index variables.

Figure 5.16 and Figure 5.17 on page 168 display the typing rules for $\mathrm{DML}_0^{\Pi}(C)$. The judgment $p \downarrow \tau \rhd (\phi; \Gamma)$ defined in Figure 5.16 on the next page is used for typing pattern matching over an expression of type τ: Type information about variables and index variables occurring in pattern p is gathered. Figure 5.17 on page 168 defines the "top-level" typing judgment $\phi; \Gamma \vdash e : \tau$ for $\mathrm{DML}_0^{\Pi}(C)$. The rule for type equality ty-eq uses a judgment $\phi \models \tau_1 \equiv \tau_2$ that is defined as the congruent extension of $\phi \models i = j$ from index objects to types. The rules ty-cons-wo, ty-cons-w and ty-iapp, for constructors and application to an index object, respectively, examine whether index objects are indeed of the required sort; when typing a constructor, the judgment for establishing type soundness of index substitutions is used to account for possible sequential dependencies among the sorts pertaining to the arguments of the constructor.

The following theorem [12, Chapter 4] shows that types are preserved under a type-sound substitution.

$$\frac{}{\phi \vdash [] : \cdot}\ \textsf{(isubst-empty)}$$

$$\frac{\phi \vdash \theta : \phi' \quad \phi \vdash i : \gamma[\theta]}{\phi \vdash \theta[a \mapsto i] : \phi', a : \gamma}\ \textsf{(isubst-ivar)}$$

$$\frac{\phi \vdash \theta : \phi' \quad \phi \models P[\theta]}{\phi \vdash \theta : \phi', P}\ \textsf{(subst-iprop)}$$

a: Type-soundness of an index substitution

$$\frac{}{\phi; \Gamma \vdash [] : (\cdot; \cdot)}\ \textsf{(subst-empty)}$$

$$\frac{\phi; \Gamma \vdash \theta : (\phi'; \Gamma') \quad \phi; \Gamma \vdash e : \tau[\theta]}{\phi; \Gamma \vdash \theta[x \mapsto e] : (\phi'; \Gamma', x : \tau)}\ \textsf{(subst-var)}$$

$$\frac{\phi; \Gamma \vdash \theta : (\phi'; \Gamma') \quad \phi \vdash i : \gamma[\theta]}{\phi; \Gamma \vdash \theta[a \mapsto i] : (\phi', a : \gamma; \Gamma')}\ \textsf{(subst-ivar)}$$

$$\frac{\phi; \Gamma \vdash \theta : (\phi'; \Gamma') \quad \phi \models P[\theta]}{\phi; \Gamma \vdash \theta : (\phi', P; \Gamma')}\ \textsf{(subst-iprop)}$$

b: Type-soundness of a substitution (general)

Figure 5.15: Type-soundness of a substitution

$$\frac{}{x \downarrow \tau \ \rhd\ (\cdot; x : \tau)}\ \textsf{(pat-var)} \qquad \frac{}{\langle\rangle \downarrow \mathbf{1} \ \rhd\ (\cdot; \cdot)}\ \textsf{(pat-unit)}$$

$$\frac{p_1 \downarrow \tau_1 \ \rhd\ (\phi_1; \Gamma_1) \quad p_2 \downarrow \tau_2 \ \rhd\ (\phi_2; \Gamma_2)}{\langle p_1, p_2 \rangle \downarrow \tau_1 \times \tau_2 \ \rhd\ (\phi_1, \phi_2; \Gamma_1, \Gamma_2)}\ \textsf{(pat-prod)}$$

$$\frac{\mathcal{S}(c) = \Pi\, a_1 : \gamma_1 \ldots \Pi\, a_k : \gamma_k \,.\, \delta(i)}{c[a_1] \ldots [a_k] \downarrow \delta(j) \ \rhd\ (a_1 : \gamma_1, \ldots, a_k : \gamma_k, i = j; \cdot)}\ \textsf{(pat-cons-wo)}$$

$$\frac{\mathcal{S}(c) = \Pi\, a_1 : \gamma_1 \ldots \Pi\, a_k : \gamma_k \,.\, \tau \to \delta(i) \quad p \downarrow \tau \ \rhd\ (\phi; \Gamma)}{c[a_1] \ldots [a_k](p) \downarrow \delta(j) \ \rhd\ (a_1 : \gamma_1, \ldots, a_k : \gamma_k, i = j, \phi; \Gamma)}\ \textsf{(pat-cons-w)}$$

Figure 5.16: Typing rules for patterns in $\mathrm{DML}_0^{\Pi}(C)$

$$\frac{\phi;\Gamma \vdash e : \tau_1 \quad \phi \models \tau_1 \equiv \tau_2}{\phi;\Gamma \vdash e : \tau_2}\ \textsf{(ty-eq)} \qquad \frac{\Gamma(x) = \tau}{\phi;\Gamma \vdash x : \tau}\ \textsf{(ty-var)}$$

$$\frac{\begin{array}{c}\mathcal{S}(c) = \Pi\, a_1 : \gamma_1 \ldots \Pi\, a_k : \gamma_k \,.\, \delta(i) \\ \phi \vdash [a_1, \ldots, a_k \mapsto i_1, \ldots, i_k] : (a_1 : \gamma_1, \ldots, a_k : \gamma_k)\end{array}}{\phi;\Gamma \vdash c[i_1] \ldots [i_k] : \delta(i[a_1, \ldots, a_k \mapsto i_1, \ldots, i_k])}\ \textsf{(ty-cons-wo)}$$

$$\frac{\begin{array}{c}\mathcal{S}(c) = \Pi\, a_1 : \gamma_1 \ldots \Pi\, a_k : \gamma_k \,.\, \tau \to \delta(i) \\ \phi \vdash [a_1, \ldots, a_k \mapsto i_1, \ldots, i_k] : (a_1 : \gamma_1, \ldots, a_k : \gamma_k) \\ \phi;\Gamma \vdash e : \tau[a_1, \ldots, a_k \mapsto i_1, \ldots, i_k]\end{array}}{\phi;\Gamma \vdash c[i_1] \ldots [i_k](e) : \delta(i[a_1, \ldots, a_k \mapsto i_1, \ldots, i_k])}\ \textsf{(ty-cons-w)}$$

$$\frac{}{\phi;\Gamma \vdash \langle\rangle : \mathbf{1}}\ \textsf{(ty-unit)} \qquad \frac{\phi;\Gamma \vdash e_1 : \tau_1 \quad \phi;\Gamma \vdash e_2 : \tau_2}{\phi;\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2}\ \textsf{(ty-prod)}$$

$$\frac{\begin{array}{c}\phi;\Gamma \vdash e : \tau' \\ \phi;\Gamma \vdash (p_1 \Rightarrow e_1) : \tau' \Rightarrow \tau \\ \vdots \\ \phi;\Gamma \vdash (p_k \Rightarrow e_k) : \tau' \Rightarrow \tau\end{array}}{\phi;\Gamma \vdash (\mathbf{case}\ e\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k) : \tau}\ \textsf{(ty-case)}$$

$$\frac{p \downarrow \tau' \ \rhd\ (\phi';\Gamma') \qquad \phi, \phi';\Gamma, \Gamma' \vdash e : \tau}{\phi;\Gamma \vdash (p \Rightarrow e) : \tau' \Rightarrow \tau}\ \textsf{(ty-branch)}$$

$$\frac{\phi, a : \gamma;\Gamma \vdash e : \tau}{\phi;\Gamma \vdash (\lambda a : \gamma \,.\, e) : \Pi\, a : \gamma \,.\, \tau}\ \textsf{(ty-ilam)}$$

$$\frac{\phi;\Gamma \vdash e : \Pi\, a : \gamma \,.\, \tau \qquad \phi \vdash i : \gamma}{\phi;\Gamma \vdash e[i] : \tau[a \mapsto i]}\ \textsf{(ty-iapp)}$$

$$\frac{\phi;\Gamma, x : \tau_1 \vdash e : \tau_2}{\phi;\Gamma \vdash (\mathbf{lam}\ x : \tau_1 \,.\, e) : \tau_1 \to \tau_2}\ \textsf{(ty-lam)}$$

$$\frac{\phi;\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \phi;\Gamma \vdash e_2 : \tau_1}{\phi;\Gamma \vdash e_1(e_2) : \tau_2}\ \textsf{(ty-app)}$$

$$\frac{\phi;\Gamma \vdash e_1 : \tau_1 \quad \phi;\Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\phi;\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : \tau_2}\ \textsf{(ty-let)}$$

$$\frac{\phi;\Gamma, f : \tau \vdash e : \tau}{\phi;\Gamma \vdash (\mathbf{fix}\ f : \tau . e) : \tau}\ \textsf{(ty-fix)}$$

Figure 5.17: Typing rules for $\mathrm{DML}_0^{\Pi}(C)$

Theorem 5.6 (Substitution) *If $\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau$ and $\phi; \Gamma \vdash \theta : (\phi'; \Gamma')$ are derivable, then $\phi; \Gamma \vdash e[\theta] : \tau[\theta]$ is derivable.*

5.A.2 DML semantics

Figure 5.18 on the next page describes a natural semantics for $\mathrm{DML}_0^{\Pi}(C)$: $e \longrightarrow v$ means that e reduces to a value v, where

$$v ::= c[i_1] \ldots [i_k] \mid c[i_1] \ldots [i_k](v) \mid \langle\rangle \mid \langle v_1, v_2 \rangle \mid (\mathbf{lam}\ x : \tau\,.\,e) \mid (\lambda a : \gamma\,.\,v).$$

Notice that type indices are never evaluated. The language design decision is that there is no direct interaction between indices and code execution; type indices are used only for type-checking.

The rule that describes the semantics of a case expression makes use of a judgment $\mathbf{match}(v, p) \Longrightarrow \theta$ defined in Figure 5.19 on page 171: If a value v matches a pattern p, a substitution for the free variables in p is returned. Notice that the semantics of case expressions is nondeterministic: an arbitrary matching arm is picked.

Xi [12, Chapter 4] proves the following theorem connecting the type system and the semantics:

Theorem 5.7 (Relating types and semantics) *1. Assume that there is no $a \in \mathbf{dom}(\phi)$ that occurs in pattern p. If $\phi; \Gamma \vdash v : \tau$, $p \downarrow \tau \rhd (\phi'; \Gamma')$ and $\mathbf{match}(p, v) \Longrightarrow \theta$, then $\phi; \Gamma \vdash \theta : (\phi'; \Gamma')$ is derivable.*

2. Given e,v in $DML_0^{\Pi}(C)$ such that $e \longrightarrow v$ is derivable. If $\phi; \Gamma \vdash e : \tau$ is derivable, then $\phi; \Gamma \vdash v : \tau$ is derivable.

$$\frac{}{c[i_1]\ldots[i_k] \longrightarrow c[i_1]\ldots[i_k]}\ \textsf{(ev-cons-wo)}$$

$$\frac{e \longrightarrow v}{c[i_1]\ldots[i_k](e) \longrightarrow c[i_1]\ldots[i_k](v)}\ \textsf{(ev-cons-w)}$$

$$\frac{e_1 \longrightarrow v_1 \quad e_2 \longrightarrow v_2}{\langle e_1, e_2\rangle \longrightarrow \langle v_1, v_2\rangle}\ \textsf{(ev-prod)}$$

$$\frac{e_0 \longrightarrow v_0 \quad \mathbf{match}(v_0, p_l) \Longrightarrow \theta \text{ for some } 1 \le l \le k \quad e_l[\theta] \longrightarrow v}{\mathbf{case}\ e_0\ \mathbf{of}\ (p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k) \longrightarrow v}\ \textsf{(ev-case)}$$

$$\frac{e \longrightarrow v}{(\lambda a : \gamma\,.\,e) \longrightarrow (\lambda a : \gamma\,.\,v)}\ \textsf{(ev-ilam)}$$

$$\frac{e \longrightarrow (\lambda a : \gamma\,.\,v)}{e[i] \longrightarrow v[a \mapsto i]}\ \textsf{(ev-iapp)}$$

$$\frac{}{(\mathbf{lam}\ x : \tau\,.\,e) \longrightarrow (\mathbf{lam}\ x : \tau\,.\,e)}\ \textsf{(ev-lam)}$$

$$\frac{e_1 \longrightarrow (\mathbf{lam}\ x : \tau\,.\,e) \quad e_2 \longrightarrow v_2 \quad e[x \mapsto v_2] \longrightarrow v}{e_1(e_2) \longrightarrow v}\ \textsf{(ev-app)}$$

$$\frac{e_1 \longrightarrow v_1 \quad e_2[x \mapsto v_1] \longrightarrow v}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} \longrightarrow v}\ \textsf{(ev-let)}$$

$$\frac{}{(\mathbf{fix}\ f : \tau.e) \longrightarrow e[f \mapsto (\mathbf{fix}\ f : \tau.e)]}\ \textsf{(ev-fix)}$$

Figure 5.18: Natural Semantics of $\mathrm{DML}_0^{\Pi}(C)$

$$\frac{}{\mathbf{match}(x, v) \Longrightarrow [x \mapsto v]}\ \text{(mat-var)}$$

$$\frac{}{\mathbf{match}(\langle\rangle, \langle\rangle) \Longrightarrow []}\ \text{(mat-unit)}$$

$$\frac{\mathbf{match}(p_1, v_1) \Longrightarrow \theta_1 \quad \mathbf{match}(p_2, v_2) \Longrightarrow \theta_2}{\mathbf{match}(\langle p_1, p_2\rangle, \langle v_1, v_2\rangle) \Longrightarrow \theta_1\theta_2}\ \text{(mat-prod)}$$

$$\frac{}{\mathbf{match}(c[a_1]\ldots[a_k], c[i_1]\ldots[i_k]) \Longrightarrow [a_1 \mapsto i_1, \ldots, a_k \mapsto i_k]}\ \text{(mat-cons-wo)}$$

$$\frac{\mathbf{match}(p, v) \Longrightarrow \theta}{\mathbf{match}(c[a_1]\ldots[a_k](p), c[i_1]\ldots[i_k](v)) \Longrightarrow \theta[a_1 \mapsto i_1, \ldots, a_k \mapsto i_k]}\ \text{(mat-cons-w)}$$

Figure 5.19: Semantics of pattern matching in $\mathrm{DML}_0^{\Pi}(C)$

5.B Formal development

5.B.1 A modified semantics a first-order fragment of DML

Figure 5.20 displays a modified semantics in which an environment of function definitions is maintained. Rule ev-mod-fundef shows how the environment is built up from function definitions, rule ev-mod-fapp shows how the environment is used. The judgment $\mathbf{match}(v, p) \Longrightarrow \theta$ in rule ev-mod-case is defined as in the standard semantics (see Figure 5.19 on the preceding page): If value v can be matched against pattern p, the corresponding substitution θ of the free variables in p is returned.

We present a proof of Theorem 5.1 on page 153, which states that the original

$$\frac{}{c[i_1]\ldots[i_k] \longrightarrow_\Theta c[i_1]\ldots[i_k]}\ \text{(ev-mod-cons-wo)}$$

$$\frac{e \longrightarrow_\Theta v}{c[i_1]\ldots[i_k](e) \longrightarrow_\Theta c[i_1]\ldots[i_k](v)}\ \text{(ev-mod-cons-w)}$$

$$\frac{e_1 \longrightarrow_\Theta v_1 \quad e_2 \longrightarrow_\Theta v_2}{\langle e_1, e_2\rangle \longrightarrow_\Theta \langle v_1, v_2\rangle}\ \text{(ev-mod-prod)}$$

$$\frac{\begin{array}{l} e_0 \longrightarrow_\Theta v_0 \\ \mathbf{match}(v_0, p_l) \Longrightarrow \theta \text{ for some } 1 \le l \le k \\ e_l[\theta] \longrightarrow_\Theta v \end{array}}{\mathbf{case}\ e_0\ \mathbf{of}\ (p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k) \longrightarrow_\Theta v}\ \text{(ev-mod-case)}$$

$$\frac{e_1 \longrightarrow_\Theta v_1 \quad e_2[x \mapsto v_1] \longrightarrow_\Theta v}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} \longrightarrow_\Theta v}\ \text{(ev-mod-let)}$$

$$\frac{e \longrightarrow_{\Theta[F \mapsto e_F]} v}{\mathbf{let}\ F = \mathbf{fix}\ F : \tau . e_F\ \mathbf{in}\ e\ \mathbf{end} \longrightarrow_\Theta v}\ \text{(ev-mod-fundef)}$$

$$\frac{\begin{array}{l} e \longrightarrow_\Theta v' \\ \Theta(F) = \lambda \vec{a} : \vec{\gamma} \,.\, \mathbf{lam}\ x : \rho \,.\, body \\ body[\vec{a} \mapsto \vec{\imath}][x \mapsto v'] \longrightarrow_\Theta v \end{array}}{F[\vec{\imath}](e) \longrightarrow_\Theta v}\ \text{(ev-mod-fapp)}$$

Figure 5.20: Natural Semantics of first-order fragment of DML

semantics of $\mathrm{DML}_0^{\Pi}(C)$ from Appendix 5.A.2 and the modified semantics of Figure 5.20 on the facing page are equivalent.

Given a program p of the form described in Figure 5.8 on page 153, the modified semantics builds up an environment from the function definitions in p (rule ev-mod-fundef), while the original semantics accumulates a substitution (rule ev-let). The judgment $\vdash \Theta : \Gamma$ (Figure 5.21) is used to establish the well-formedness of an environment Θ and to type the functions contained in it.

$$\frac{}{\vdash [] : \cdot}\ \textsf{(ty-env-nil)}$$

$$\frac{\vdash \Theta : \Gamma \quad \cdot; \Gamma, F : \tau \vdash \lambda\vec{a} : \vec{\gamma}\,.\,\mathbf{lam}\ x : \rho_F\,.\,body_F : \tau}{\vdash \Theta[F \mapsto \lambda\vec{a} : \vec{\gamma}\,.\,\mathbf{lam}\ x : \rho_F\,.\,body_F] : \Gamma, F : \tau}\ \textsf{(ty-env-cons)}$$

Figure 5.21: Typing an environment

We further use a mapping $(\cdot)^\circ$ to relate the environment of the modified semantics with the substitution of the original semantics. Given an environment $\vdash \Theta : \Gamma$, the substitution Θ° is defined as follows:

$$\begin{array}{rcl} ([])^\circ & = & [] \\ (\Theta[F \mapsto e])^\circ & = & [F \mapsto \mathbf{fix}\ F : \Gamma(F).e] \circ \Theta^\circ \end{array}$$

The following lemma relates the modified semantics with the original semantics.

Lemma 5.8 *Let $\vdash \Theta : \Gamma_F$ and let e be a body expression such that $\phi; \Gamma_F, \Gamma \vdash e : \rho$ is derivable. Then $e[\Theta^\circ] \longrightarrow v$ iff $e \longrightarrow_\Theta v$.*

Proof: The proof of the lemma is conducted by structural induction over the derivation $e[\Theta^\circ] \longrightarrow v$ (proving the implication from left to right) and $e \longrightarrow_\Theta v$ (right to left).

We show the implication from left to right, examining the last rule in a derivation of $e[\Theta^\circ] \longrightarrow v$. For rules ev-cons-wo, ev-cons-w, ev-prod, ev-case and ev-let, the lemma follows immediately by induction hypothesis. For rules ev-ilam, ev-iapp, ev-app and ev-fix observe, that because of the restricted shape of e (see Figure 5.8 on page 153), only rule ev-ilam can occur as the last rule in a derivation $e[\Theta^\circ] \longrightarrow v$, namely for $e = F[\vec{\imath}](e')$. Assume therefore that $(F[\vec{\imath}](e'))[\Theta^\circ] \longrightarrow v$ is derivable; an analysis of the shape of the corresponding derivation (see Figure 5.18 on page 170 for the rules) shows that for some value v_1 there are derivations of $e'[\Theta^\circ] \longrightarrow v_1$ and $body[\vec{a} \mapsto \vec{\imath}][x \mapsto v_1][\Theta^\circ] \longrightarrow v$. Using the induction hypothesis, we can derive that $body[\vec{a} \mapsto \vec{\imath}][x \mapsto v_1] \longrightarrow_\Theta v$,

from which it follows easily with rule ev-mod-fapp (Figure 5.20 on page 172) that $F[\vec{i}](e') \longrightarrow_\Theta v$.

The other direction of the implication follows similarly. □

We are now in a position to prove Theorem 5.1 on page 153.

Proof: Let program p be of the form given in Figure 5.8 on page 153 with function definitions $F_l = \mathbf{fix}\ F_l : \tau_l.e_l$ for $0 \leq l \leq k$ and program body e. With the typing rules ty-let and ty-fix (Figure 5.17 on page 168) and the rules from Figure 5.21 on the previous page a straightforward induction over the number of function definitions shows that

$$\vdash (F_1 : \tau_1, \ldots, F_k : \tau_k) : \overbrace{[F_1 \mapsto e_1, \ldots, F_k \mapsto e_k]}^{=:\Theta}.$$

Further, examining the modified and original semantics, we see:

$$\mathsf{p} \longrightarrow v \quad \text{iff} \quad e[\Theta^\circ] \longrightarrow v$$
$$\mathsf{p} \longrightarrow_{[]} v \quad \text{iff} \quad e \longrightarrow_\Theta v$$

With Lemma 5.8 on the preceding page it follows that $\mathsf{p} \longrightarrow v$ iff $\mathsf{p} \longrightarrow_{[]} v$. □

5.B.2 The monadic translation

We present a proof of Theorem 5.2 on page 155, which states that the monadic translation preserves types and semantics.

The monadic translation preserves types

The following Lemma expresses the validity of the typing rules given for $\mathbf{val}^\mathsf{C}$, $\mathbf{let}^\mathsf{C}$ and **cost** in Section 5.4.2 with respect to the expansion of these constructs as given in Figure 5.10 on page 156.

Lemma 5.9 *The typing rules for* $\mathbf{val}^\mathsf{C}$, $\mathbf{let}^\mathsf{C}$ *and* **cost** *as given in Section 5.4.2 are admissible (assuming that* $\mathbf{val}^\mathsf{C}$, $\mathbf{let}^\mathsf{C}$ *and* **cost** *are expanded as described in Figure 5.10 on page 156).*

Proof: Straightforward by constructing the corresponding type derivations. □

The first part of Theorem 5.2 on page 155 follows immediately from the following lemma.

Lemma 5.10 *For an expression* e*, if* $\phi; \Gamma \vdash e{:}\rho$ *is derivable, then* $\phi; \Gamma^* \vdash e^*{:}\mathsf{C}\,\rho$ *is derivable, where* Γ^* *results from* Γ *by wrapping the result type of every function declared in* Γ *with* C*.*

Proof: The proof is conducted by structural induction on the derivation of $\phi; \Gamma \vdash e : \rho$ (typing rules in Figure 5.17 on page 168). If the last rule of the derivation is (ty-eq), then the lemma follows immediately by induction hypothesis. Otherwise, because the remaining rules are syntax directed, we proceed by examining all possible expressions e.

We present one interesting case, namely a function call $\phi; \Gamma \vdash F[\vec{i}](e) : \rho$. Examining the possible derivations, we see that there exists ρ_1 such that $\phi; \Gamma \vdash F\ [\vec{i}] : \rho_1 \to \rho$ and $\phi; \Gamma \vdash e : \rho_1$ are derivable. The monadic translation of $F[\vec{i}](e)$ is

$$\begin{array}{l}\textbf{let}^{\mathsf{C}}\ x = e^* \\ \textbf{in cost}_{\mathbf{c}_c}\ (F[\vec{i}](x))\ \textbf{end}\end{array}$$

Because of Lemma 5.9 on the facing page, rule ty-monadic-let (Section 5.4.2) is admissible, hence we need to find derivations of $\phi; \Gamma \vdash e^* : \mathsf{C}\,\rho_1$ and $\phi; \Gamma^*, x : \rho_1 \vdash \textbf{cost}_{c_F}\ (F[\vec{i}](x)) : \mathsf{C}\,\rho$. The former follows by induction hypothesis, the latter is easily derived using rule ty-monadic-cost and the fact that $\phi; \Gamma \vdash F\ [\vec{i}] : \rho_1 \to \rho$ is derivable. □

The monadic translation preserves semantics

In order to prove the second part of Theorem 5.2 on page 155, namely that the monadic translation preserves semantics, we need the following lemma:

Lemma 5.11 *Let $\vdash \Theta : \Gamma_F$ and $\phi; \Gamma_F, \Gamma \vdash e : \rho$ be derivable. Then $e \longrightarrow_\Theta v$ iff there exists a $z \in \mathbf{C}$ such that $e^* \longrightarrow_{\Theta^*} \langle v, z\rangle$ is derivable (where $\Theta^*(F) = \Theta(F)^*$ for all $F \in \mathbf{dom}(\Theta)$).*

Proof: The proof is conducted by structural induction over $e \longrightarrow_\Theta v$ (proving the implication from left to right) and over $e^* \longrightarrow_{\Theta^*} \langle v, z\rangle$ (right to left).

We show the case of a function call $F[\vec{i}](e)$. Assume that there exists a derivation of $F[\vec{i}](e) \longrightarrow_\Theta v$. Rule inversion with ev-mod-fapp (Figure 5.20 on page 172) shows that there exists a value v' such that $e \longrightarrow_\Theta v'$ and $body[\vec{a} \mapsto \vec{i}][x_1 \mapsto v'] \longrightarrow_\Theta v$ are derivable, where $\Theta(F) = \lambda\vec{a} : \vec{\gamma}\,.\,\textbf{lam}\ x : \rho\,.\,body$. Consider now the translated term $(F[\vec{i}](e))^*$ where the monadic constructs have been expanded as shown in Figure 5.10 on page 156:

$$\begin{array}{l}\textbf{case}\ e^*\ \textbf{of} \\ \quad \langle x_1, z_1\rangle \Rightarrow\ \textbf{case}\ (\textbf{case}\ F[\vec{i}](x_1)\ \textbf{of}\ \langle x_3, z_3\rangle \Rightarrow \langle x_3, z_3 + c_F\rangle)\ \textbf{of} \\ \qquad\qquad \langle x_2, z_2\rangle \Rightarrow\ \langle x_2, z_1 + z_2\rangle\end{array}$$

By induction hypothesis, we know that there is a z_1' such that $e^* \longrightarrow_{\Theta^*} \langle v', z_1'\rangle$; with ev-mod-case it follows that we need to show a derivation of $F[\vec{i}](v') \longrightarrow_{\Theta^*} \langle v, z_3'\rangle$ for some $z_3' \in \mathbf{C}$. Such a derivation can be constructed using ev-mod-fapp and a derivation $body^*[\vec{a} \mapsto \vec{i}][x_1 \mapsto v'] \longrightarrow_{\Theta^*} \langle v, z_3'\rangle$, which exists by induction hypothesis. □

We are now in a position to prove the second part of Theorem 5.2 on page 155:

Proof: Let program p be of the form given in Figure 5.8 on page 153 with function definitions $F_l = \mathbf{fix}\ F_l : \tau_l . e_l$ for $0 \leq l \leq k$ and program body e. Let

$$\begin{array}{rcl} \Gamma & := & F_1 : \tau_1, \ldots, F_k : \tau_k \\ \Theta & := & [F_1 \mapsto e_1, \ldots, F_k \mapsto e_k] \end{array}$$

Inspecting the type derivation of p it is easy to see that $\vdash \Theta : \Gamma$ and $\cdot; \Gamma \vdash e : \rho$ are derivable. From the semantics we know that $p \longrightarrow_{[]} v$ and $p^* \longrightarrow_{[]} \langle v', z \rangle$ iff $e \longrightarrow_\Theta v$ and $e^* \longrightarrow_{\Theta^*} \langle v', z \rangle$, respectively. With Lemma 5.11 on the preceding page we have that $e \longrightarrow_\Theta v$ iff $e^* \longrightarrow_{\Theta^*} \langle v, z \rangle$, which concludes the proof. □

5.B.3 Extraction of recurrence equations—preliminaries

An erasure from index sorts to index types

The erasure $\widetilde{\cdot}$ maps an index sort to its associated index type by removing all constraint-related information. It is defined as follows:

$$\begin{array}{rcl} \widetilde{\mathbb{N}} & = & \mathbb{N} \\ \widetilde{\mathbf{1}} & = & \mathbf{1} \\ \widetilde{\gamma_1 \times \gamma_2} & = & \widetilde{\gamma_1} \times \widetilde{\gamma_2} \\ \widetilde{\{a : \gamma \mid P\}} & = & \widetilde{\gamma} \end{array}$$

For a context Γ that maps function names to types we define $\widetilde{\Gamma}$ such that

$$\widetilde{\Gamma}(F^c) = \widetilde{\gamma_0} \to \widetilde{\gamma_1} \to \ldots \widetilde{\gamma_k} \to \mathbf{C}$$

if

$$\Gamma(F) = \Pi\, a_0 : \gamma_0 \,.\, \Pi\, a_1 : \gamma_1 \ldots \Pi\, a_k : \gamma_k \,.\, \rho_1 \to \rho_2 .$$

For an index context ϕ we define $\widetilde{\phi}$ as follows:

$$\begin{array}{rcl} \widetilde{\cdot} & = & \cdot \\ \widetilde{\phi, a : \gamma} & = & \widetilde{\phi}, a : \widetilde{\gamma} \\ \widetilde{\phi, P} & = & \widetilde{\phi} \end{array}$$

The following lemma shows that the erasure preserves type soundness of index substitutions.

Lemma 5.12 *If $\phi \vdash \theta : \phi'$ is derivable, then $\widetilde{\phi} \vdash \theta : \widetilde{\phi'}$ is derivable.*

Proof: With a straightforward induction over the structure of sort γ, it is easy to show that if $\phi \vdash i : \gamma$ is derivable, then $\widetilde{\phi} \vdash i : \widetilde{\gamma}$ is derivable. Using this fact, one then can show the lemma with a straightforward induction over the length of ϕ'. □

Flattening index contexts into constraints

The algorithm for extracting cost recurrences uses a function $\mathcal{C}$ to rewrite an index context into a conjunctive constraint by flattening subsort definitions. A declaration $i : \{k : \gamma \mid P\}$ is rewritten with the conjunction of (1) the constraints imposed by the declaration $i : \gamma$ and (2) the index proposition $P[k \mapsto i]$. The resulting constraint $\Phi := \mathcal{C}(\phi)$ is a quantifier-free conjunction of equality constraints and index propositions:[7]

$$\begin{array}{lcl}
\mathcal{C}(\cdot) & = & \top \\
\mathcal{C}(\phi, i : \mathbf{1}) & = & \mathcal{C}(\phi) \\
\mathcal{C}(\phi, i : \mathbb{N}) & = & \mathcal{C}(\phi) \\
\mathcal{C}(\phi, i : \gamma_1 \times \gamma_2) & = & \mathcal{C}(\phi, \mathbf{fst}(i) : \gamma_1, \mathbf{snd}(i) : \gamma_2) \\
\mathcal{C}(\phi, i : \{k : \gamma \mid P\}) & = & \mathcal{C}(\phi, i : \gamma) \wedge P[k \mapsto i] \\
\mathcal{C}(\phi, P) & = & \mathcal{C}(\phi) \wedge P \\
\mathcal{C}(\phi, i = j) & = & \mathcal{C}(\phi) \wedge (i = j)
\end{array}$$

$\mathcal{C}$ is well-defined: Assuming a straightforward weight-function for sort definitions, a termination order for $\mathcal{C}(\phi)$ can be given as the lexicographic order of the sum of the weights of all sort definitions in ϕ and the length of ϕ.

The following lemma expresses a useful correspondence between ϕ and $\mathcal{C}(\phi)$:

Lemma 5.13 *Let ϕ and ϕ' be index contexts and θ an index substitution such that $\mathbf{dom}(\theta) = \mathbf{dom}(\phi')$. Then*

$$\phi \vdash \theta : \phi' \quad \textit{iff} \quad \phi \models \mathcal{C}(\phi')[\theta].$$

Proof: We conduct the proof by structural induction over ϕ. In the base case, $\phi = \cdot$, we have $\phi \vdash [] : \cdot$ and $\phi \models \top[\theta]$ for all substitutions θ. Both directions follow immediately (using $\mathbf{dom}(\theta) = \mathbf{dom}(\phi')$ from right to left).

A non-empty index context has form $\phi, a : \gamma$ or ϕ, P. We examine the first of these cases, the second case follows by similar reasoning.

We have to show that

$$\phi \vdash \theta[a \mapsto i] : \phi', a : \gamma \quad \text{iff} \quad \phi \models \mathcal{C}(\phi', a : \gamma)[\theta[a \mapsto i]],$$

which is equivalent to

$$(\phi \vdash \theta : \phi' \text{ and } \phi \vdash i : \gamma[\theta]) \quad \text{iff} \quad (\phi \models \mathcal{C}(\phi')[\theta] \text{ and } \phi \models \mathcal{C}(a : \gamma)[\theta[a \mapsto i]]).$$

Assuming that for all i and γ

$$\phi \vdash i : \gamma \quad \text{iff} \quad \phi \models \mathcal{C}(a : \gamma)[a \mapsto i], \tag{*}$$

we can conclude the proof using the induction hypothesis.

[7] Because of the way $\mathcal{C}$ handles product sorts, it actually is defined on contexts that assign sorts to index objects rather than only index variables.

It remains to show Equation (*). We use induction over the structure of γ and demonstrate the case of a sort definition of form $\{k : \gamma \mid P\}$: We have to show that

$$\phi \vdash i : \{k : \gamma \mid P\} \quad \text{iff} \quad \phi \models \mathcal{C}(a : \{k : \gamma \mid P\})[a \mapsto i]$$

Assuming the left-hand side, it follows with rule-inversion that $\phi \vdash i : \gamma$ and $\phi \models P[k \mapsto i]$. Using the induction hypothesis, we further can derive that $\phi \models \mathcal{C}(a : \gamma)[a \mapsto i]$, so obviously $\phi \models (\mathcal{C}(a : \gamma \wedge P[k \mapsto a]))[a \mapsto i]$, which is equivalent to $\phi \models (\mathcal{C}(a : \{k : \gamma \mid P\}))[a \mapsto i]$. The direction from right to left follows with similar reasoning steps. □

5.B.4 Extraction of recurrence equations—correctness

In this section, we present a proof of the correctness of the extraction algorithm for recurrence equations, as stated in Theorem 5.4 on page 163. As the theorem is in three parts, we divide the proof into three parts.

The result of extraction is well-typed

The first part of Theorem 5.4 on page 163 follows directly from a lemma that relates the type derivation of a function body to the type derivation of the extracted recurrence-equation term.

Lemma 5.14 *If $\phi; \Gamma \vdash e : \rho \blacktriangleright t$ is derivable for an expression e, then $\widetilde{\phi}; \widetilde{\Gamma} \vdash t : \mathbf{C}$ is derivable.*

Proof: The proof is conducted by structural induction over the type derivation of e. In case the last rule of the type derivation is ty-eq, then the lemma follows by induction hypothesis. Because the remaining typing rules for $\mathrm{DML}_0^{\Pi}(C)$ are syntax directed, we proceed by examining the different syntactic forms of e as given by the grammar in Figure 5.8 on page 153. We show the case of a function call $F\ [i_0] \ldots [i_k]\ e$; all other cases can be shown in a similar way.

For a function call $F[i_0] \ldots [i_k](e)$, the extracted recurrence-equation term is $t := t' + \mathbf{c}_F + (F^c\ i_0 \ldots i_k)$ where t' has been extracted from e. The last rule of the type derivation must be ty-app, i.e., for some ρ_1, ρ_2 there exist derivations

$$\phi; \Gamma \vdash e : \rho_1 \tag{1}$$

$$\phi; \Gamma \vdash F[i_0] \ldots [i_k] : \rho_1 \to \rho_2 \tag{2}$$

Analyzing the shape of a type derivation for t, we see that we need to show

$$\widetilde{\phi}; \widetilde{\Gamma} \vdash t' : \mathbf{C} \tag{1'}$$

$$\widetilde{\phi} \vdash i_0 : \widetilde{\gamma_0} \quad \ldots \quad \widetilde{\phi} \vdash i_k : \widetilde{\gamma_k}, \tag{2'}$$

assuming that $\Gamma(F) = \Pi\, a_0 : \gamma_0 \ldots \Pi\, a_k : \gamma_k \,.\, \rho_1' \to \rho_2'$. From (1), we can show (1′) using the induction hypothesis. From (2), we can derive (2′): With a straightforward induction on k one can show that (2) implies

$$\phi \vdash [a_0 \mapsto i_0, \ldots, a_k \mapsto i_k] : (a_0 : \gamma_0, \ldots, a_k : \gamma_k).$$

With Lemma 5.12 on page 176, it follows that

$$\widetilde{\phi} \vdash [a_0 \mapsto i_0, \ldots, a_k \mapsto i_k] : (a_0 : \widetilde{\gamma_0}, \ldots, a_k : \widetilde{\gamma_k}),$$

which, because there are no dependencies between two index types $\widetilde{\gamma_l}$ and $\widetilde{\gamma_{l'}}$, implies (2′). □

The result of extraction is a cost bound

We need to show that the cost of executing $F\ [\vec{i}]\ v$ for some user-defined function F and some value v is bounded by $F^c\ \vec{i}$, where F^c is the cost recurrence extracted for F. We start with Lemma 5.15, which says that when extracting a recurrence-equation term t from a DML expression e, then t defines a bound for e under the assumption that all calls to user-defined functions in e are bounded correctly by the corresponding calls to F^c in t.

Lemma 5.15 *Let $\phi; \Gamma_F, \Gamma \vdash e : \rho \blacktriangleright t$ be derivable. Let θ be a substitution such that θ_ϕ substitutes index variables for ground terms, θ_Γ substitutes (normal) variables for values, and $\cdot; \Gamma_F \vdash \theta : (\phi; \Gamma)$ is derivable, i.e., θ is type-sound with respect to ϕ and Γ. Let Θ be an environment such that $\vdash \Theta : \Gamma_F$ is derivable. Let Ψ be a mapping from $\mathbf{dom}(\Gamma_F)$ to functions such that*

1. *if $\Gamma_F(F) = \Pi\, a_0 : \gamma_0 \ldots \Pi\, a_l : \gamma_l \,.\, \rho_1 \to \rho_2$ then*

$$\Psi(F^c) \in [\mathcal{I}[\![\widetilde{\gamma_0}]\!] \to \ldots \to \mathcal{I}[\![\widetilde{\gamma_l}]\!] \to \mathbf{C}_\perp]$$

2. *if $F[\vec{i}](u)$ type-checks under Γ_F and $\mathcal{T}[\![F^c\ \vec{i}]\!]\Psi[] = \llcorner z \lrcorner$ then*

$$F[\vec{i}](u) \longrightarrow_{\Theta^*} \langle u', z' \rangle \quad \text{with } z' \leq z.$$

Then if $\mathcal{T}[\![t]\!]\Psi(\mathcal{I}[\![\theta_\phi]\!]) = \llcorner z \lrcorner$, then $e^*[\theta] \longrightarrow_{\Theta^*} \langle v', z' \rangle$ with $z' \leq z$.

Proof: We use structural induction over the type derivation of e. In case the last rule of the type derivation is ty-eq, then the lemma follows by induction hypothesis. Because the remaining typing rules for $\mathrm{DML}_0^\Pi(C)$ are syntax directed, we proceed by examining the different syntactic forms of e as given by the grammar in Figure 5.8 on page 153. We show two interesting cases: function application and case expression.

For a function application $F[\vec{i}](e)$, the extracted recurrence-equation term is $t + \mathbf{c}_F + F^c\ \vec{i}$, where t has been extracted from e. We express $(F[\vec{i}](e))^*$, (see

Figure 5.9 on page 155 for the definition of the monadic translation $(\cdot)^*$) as a DML program by removing syntactic sugar introduced in the translation (see Figure 5.10 on page 156) and examine the shape of a possible derivation for

$$(F[\vec{i}](e))^*[\theta] \longrightarrow_{\Theta^*} \langle v', z' \rangle.$$

It is easy to see that such a derivation can exist only if there are $z'_F, z'_e \in \mathbf{C}$ such that $z' = z'_F + \mathbf{c}_F + z'_e$ and the following two assumptions hold:

(1) For some value v_e, there is a derivation $e^*[\theta] \longrightarrow_{\Theta^*} \langle v_e, z'_e \rangle$.

(2) There is a derivation $F\ [\vec{i}[\theta]]\ v_e \longrightarrow_{\Theta^*} \langle v', z'_F \rangle$.

Assume now that $\mathcal{T}[\![t + \mathbf{c}_F + F^c\ \vec{i}]\!]\Psi(\mathcal{I}[\![\theta_\phi]\!]) = \llcorner z \lrcorner$; using the semantics definition from Figure 5.13 on page 158 the following three facts follow easily:

(1′) For some $z_e \in \mathbf{C}$ we have $\mathcal{T}[\![t]\!]\Psi(\mathcal{I}[\![\theta_\phi]\!]) = \llcorner z_e \lrcorner$.

(2′) For some $z_F \in \mathbf{C}$, we have $\mathcal{T}[\![F^c\ \vec{i}]\!]\Psi(\mathcal{I}[\![\theta_\phi]\!]) = \llcorner z_F \lrcorner$.

(3′) We have $z = z_F + \mathbf{c}_F + z_e$.

Using the induction hypothesis and (1′), we can deduce (1) for some $z'_e \leq z_e$. From (2′), it follows by assumption that (2) holds for some $z'_F \leq z_F$; this is because $\mathcal{T}[\![F^c\ \vec{i}]\!]\Psi(\mathcal{I}[\![\theta_\phi]\!])$ is equivalent to $\mathcal{T}[\![F^c\ (\vec{i}[\theta_\phi])]\!]\Psi[]$. Finally, using (3′) we can infer that $z' \leq z$, which concludes the proof for this case.

For a case expression of form (**case** e **of** $p_0 \Rightarrow e_0 \mid \ldots p_k \Rightarrow e_k$), the extracted recurrence-equation term is

$$t + (\mathbf{cond}\ br_0 \mid \ldots \mid br_k)$$

where t is extracted from e by

$$\phi; \Gamma_F, \Gamma \vdash e : \rho' \blacktriangleright t$$

and every br_j from the type-derivation of a branch $(p_j \Rightarrow e_j)$ as

$$\Phi_{j1} \rightarrow \forall(\mathbf{dom}(\phi'_j) \backslash \mathbf{dom}(\theta_j)).\Phi_{j2} \rightarrow t[\theta_j]$$

where

$$\begin{array}{l} p \downarrow \rho' \rhd (\phi'_j; \Gamma'_j) \\ \phi, \phi'_j; \Gamma, \Gamma'_j \vdash e_j : \rho \blacktriangleright t_j \\ \Phi_{j1} = \exists(\mathbf{dom}(\phi, \phi'_j) \backslash \mathbf{var}(\rho')).\mathcal{C}(\phi, \phi'_j) \\ \theta_j = \mathit{mk\_subst}_{\mathbf{dom}(\phi'_j)}(\mathcal{C}(\phi'_j)) \\ \Phi_{j2} = \exists(\mathbf{dom}(\theta_j)).\mathcal{C}(\phi'_j) \end{array}$$

Like before, we examine the shape of a possible derivation for

$$(\mathbf{case}\ e\ \mathbf{of}\ p_0 \Rightarrow e_0 \mid \ldots p_k \Rightarrow e_k)^*[\theta] \longrightarrow_{\Theta^*} \langle v', z' \rangle,$$

and see that such a derivation can exist only if there are $z'_e, z'_b \in \mathbf{C}$ such that $z' = z'_e + z'_b$ and the following two assumptions hold:

(1) For some value v_e, there is a derivation $e^*[\theta] \longrightarrow_{\Theta^*} \langle v_e, z'_e\rangle$.

(2) For some j with $0 \leq j \leq k$ for which $\mathbf{match}(p_j, v_e) \Longrightarrow \theta'$ is derivable, there is a derivation ${e_j}^*[\theta \circ \theta'] \longrightarrow_{\Theta^*} \langle v', z'_b\rangle$.

Assume now that $\mathcal{T}[\![t + (\mathbf{cond}\ br_0 \mid \ldots \mid br_k)]\!]\Psi(\mathcal{I}[\![\theta_\phi]\!]) = \llcorner z \lrcorner$; using the semantics definition from Figure 5.13 on page 158 we can deduce the following facts:

(1′) For some $z_e \in \mathbf{C}$ we have $\mathcal{T}[\![t]\!]\Psi(\mathcal{I}[\![\theta_\phi]\!]) = \llcorner z_e \lrcorner$.

(2′) There is some $z_b \in \mathbf{C}$ such that for every branch br_j

$$\Phi_{j1} \to \forall(\mathbf{dom}(\phi'_j)\backslash\mathbf{dom}(\theta_j)).\Phi_{j2} \to t[\theta_j]$$

we have that if $\models \Phi_{j1}[\theta_\phi]$ then for all $\vec{z}$ with $\models \Phi_{j2}[\theta_\phi[\vec{a} \mapsto \vec{z}]]$ there is $z_B \leq z_b$ such that $\mathcal{T}[\![t_j[\theta_j]]\!]\Psi(\mathcal{I}[\![\theta_\phi[\vec{a} \mapsto \vec{z}]]\!]) = \llcorner z_B \lrcorner$

(3′) We have $z = z_e + z_b$.

Using the induction hypothesis and (1′), we can deduce (1) for some $z'_e \leq z_e$. In the following, we prove (2), using the induction hypothesis and (2′):

- We first show that whenever $\mathbf{match}(p_j, v_e) \Longrightarrow \theta'$ is derivable, then $\models \Phi_{j1}[\theta_\phi]$, i.e., the precondition for (2') holds.

  We examine the form of $\Phi_{j1} = \exists\vec{b}, \vec{c}.\mathcal{C}(\phi, \phi'_j)$, where $\vec{b}$ are the variables in $\mathbf{dom}(\phi'_j)$ and $\vec{c}$ are the variables in $\mathbf{dom}(\phi)\backslash\mathbf{var}(\rho')$. We need to show

  $$\models (\exists\vec{b}, \vec{c}.\mathcal{C}(\phi, \phi'_j))[\theta_\phi].$$

  In fact, θ_ϕ and ${\theta'}_\phi$ hold witnesses for $\vec{b}$ and $\vec{c}$, respectively: By assumption, we have $\cdot; \Gamma_F \vdash \theta : (\phi; \Gamma)$, which implies $\vdash \theta_\phi : \phi$. Further, with the first part of Theorem 5.7 on page 169, it follows that $\phi \vdash \theta_{j\,\phi} : \phi'$. Using Lemma 5.13 on page 177, we derive $\models \mathcal{C}(\phi)[\theta_\phi]$ and $\phi \models \mathcal{C}(\phi'_j)[{\theta'}_\phi]$. With Lemma 5.5 on page 166, we then can derive $\models (\mathcal{C}(\phi'_j)[{\theta'}_\phi])[\theta_\phi]$, so all in all we have we have $\models \mathcal{C}(\phi, \phi'_j)[{\theta'}_\phi\ \theta_\phi]$, which implies $\models (\exists\vec{b}, \vec{c}.\mathcal{C}(\phi, \phi'_j))[\theta_\phi]$.

- With similar reasoning, we show that whenever $\mathbf{match}(p_j, v_e) \Longrightarrow \theta'$ is derivable, then $\models \Phi_{j2}[\theta_\phi[\vec{a} \mapsto {\theta'}_\phi(\vec{a})]]$ (where $\vec{a} := \mathbf{dom}(\phi'_j)\backslash\mathbf{dom}(\theta_j)$) holds, so with (2′) it follows that

  $$\mathcal{T}[\![t_j[\theta_j]]\!]\Psi(\mathcal{I}[\![\theta[\vec{a} \mapsto {\theta'}_\phi(\vec{a})]]\!]) = \llcorner z_B \lrcorner \quad \text{for some } z_B \leq z_b. \qquad (*)$$

- Knowing (*), we can show (2) by induction hypothesis if

  $$\mathcal{T}[\![t_j[\theta_j]]\!]\Psi(\mathcal{I}[\![\theta_\phi[\vec{a} \mapsto {\theta'}_\phi(\vec{a})]]\!]) = \mathcal{T}[\![t_j]\!]\Psi(\mathcal{I}[\![\theta_\phi \circ {\theta'}_\phi]\!]).$$

  With a straightforward structural induction over t_j we can show that

  $$\mathcal{T}[\![t_j[\theta_j]]\!]\Psi(\mathcal{I}[\![\theta_\phi[\vec{a} \mapsto {\theta'}_\phi(\vec{a})]]\!]) = \mathcal{T}[\![t_j]\!]\Psi(\mathcal{I}[\![\theta_j \circ \theta_\phi[\vec{a} \mapsto {\theta'}_\phi(\vec{a})]]\!])$$

It remains to show that $b[\theta_j \circ \theta_\phi[\vec{a} \mapsto \theta'_\phi(\vec{a})]] = b[\theta_\phi \circ \theta'_\phi]$ for all $b \in \mathbf{dom}(\theta \circ \theta')$. Some reasoning about the domains of the comprised substitutions allows us to rearrange their composition and show instead that

$$b[(\theta_j \circ [\vec{a} \mapsto \theta'_\phi(\vec{a})])\theta_\phi] = b[\theta'_\phi\ \theta_\phi] \qquad (**)$$

For $b \in \vec{a}$ and $b \in \mathbf{dom}(\theta)$ there is nothing to show. Consider now $b \in \mathbf{dom}(\theta_j)$. By definition of θ_j we know that $\phi' \models b = b[\theta_j]$. Using Lemma 5.5 on page 166 on $\vdash \theta_\phi : \phi$ and $\phi \vdash \theta_{j\,\phi} : \phi'_j$ (established above), we can derive that $b[\theta_j \circ \theta'_\phi \circ \theta_\phi] = b[\theta'_\phi \circ \theta_\phi]$. Equation (**) follows with some basic reasoning about substitutions.

Now, with (3′) and the fact that $z_B \leq z_b$, it follows that $z' \leq z$, which concludes the proof for this case. □

We have shown that extracting a recurrence-equation term t from a DML expression e yields a valid bound under the assumption that we have a valid bound F^c for every user-defined function F called in e. We now show that the semantics of a recurrence equation G^c, which is based on the recurrence-equation term extracted from the body of a user-defined function G (see Section 5.4.3), indeed defines a bound for G.

Lemma 5.16
Let $(\mathbf{fix}\ G : \Pi\,\vec{a} : \vec{\gamma}\,.\,\rho_1 \to \rho_2.\mathbf{lam}\ x : \rho_1\,.\,\mathbf{case}\ x\ \mathbf{of}\ \langle x_0, \ldots, x_k\rangle \Rightarrow e)$ *be typable under* Γ_F *and let*

$$\cdot; \Gamma_F, G : \Pi\,\vec{a} : \vec{\gamma}\,.\,\rho_1 \to \rho_2, x : \rho_1, x_0 : \rho_1^0, \ldots, x_k : \rho_1^k \vdash\ e : \rho_2\ \blacktriangleright\ t$$

be derivable. Let Θ *be an environment such that* $\vdash \Theta : \Gamma_F$ *is derivable. Let* Ψ *be a mapping from* $\mathbf{dom}(\Gamma_F)$ *to functions such that*

1. *if* $\Gamma_F(F) = \Pi\,a_0 : \gamma_0 \ldots \Pi\,a_l : \gamma_l\,.\,\rho_1 \to \rho_2$ *then*

$$\Psi(F^c) \in [\mathcal{I}[\![\widetilde{\gamma}_0]\!] \to \ldots \to \mathcal{I}[\![\widetilde{\gamma}_l]\!] \to \mathbf{C}_\perp]$$

2. *if* $F[\vec{i}](u)$ *type-checks under* Γ_F *and* $\mathcal{T}[\![F^c\ \vec{i}]\!]\Psi[] = \llcorner z \lrcorner$ *then*

$$F[\vec{i}](u) \longrightarrow_{\Theta^*} \langle u', z'\rangle \quad \textit{with } z' \leq z.$$

Let further

$$\begin{array}{rcl} \varphi & := & \mathit{fix}(\lambda\mathcal{F}.\lambda\vec{n}.\mathcal{T}[\![t]\!](\Psi[G^c \mapsto \mathcal{F}])[\vec{a} \mapsto \vec{n}]) \\ \Theta_1 & := & \Theta[G \mapsto \mathbf{lam}\ x : \rho_1\,.\,\mathbf{case}\ x\ \mathbf{of}\ \langle x_0, \ldots, x_k\rangle \Rightarrow e] \end{array}$$

Then, if $G[\vec{i}](u)$ *type-checks under* $\Gamma_F, G : \Pi\,\vec{a} : \vec{\gamma}\,.\,\rho_1 \to \rho_2$ *and*

$$\mathcal{T}[\![G^c\ [\vec{i}]]\!](\Psi[G^c \mapsto \varphi])[] = \llcorner z \lrcorner,$$

we have $G[\vec{i}](u) \longrightarrow_{\Theta_1{}^*} \langle v', z'\rangle$ *with* $z' \leq z$.

Proof: We use fixed-point induction over the definition of φ. For the base case, we have

$$\mathcal{T}[\![G\ [\vec{\imath}]]\!](\Psi[G^c \mapsto \varphi])[] = \bot,$$

so the lemma is vacuously true. For the induction step, assume that for φ', if $G[\vec{\imath}](u)$ type-checks under $\Gamma_F, G : \Pi\,\vec{a} : \vec{\gamma}\,.\,\rho_1 \to \rho_2$ and

$$\mathcal{T}[\![G^c\ [\vec{\imath}]]\!](\Psi[G^c \mapsto \varphi'])[] = \llcorner z \lrcorner,$$

we have $G[\vec{\imath}](u) \longrightarrow_{\Theta_1{}^*} \langle v', z'\rangle$ with $z' \leq z$. We have to show that if

$$\mathcal{T}[\![G\ [\vec{\imath}]]\!](\Psi[G^c \mapsto \lambda\vec{n}.\mathcal{T}[\![t]\!](\Psi[G^c \mapsto \varphi'])[\vec{a} \mapsto \vec{n}]])[] = \llcorner z \lrcorner \tag{5.1}$$

then

$$G[\vec{\imath}]\ u \longrightarrow_{\Theta_1{}^*} \langle v', z'\rangle \tag{5.2}$$

with $z' \leq z$.

Using the definitions of the denotational semantics (Figure 5.13 on page 158) of recurrence equations and operational semantics of DML (Figure 5.20 on page 172), we see that Equations (5.1) and (5.2) are equivalent to

$$\mathcal{T}[\![t]\!](\Psi[G^c \mapsto \varphi'])[\vec{a} \mapsto \mathcal{I}[\![\vec{\imath}]\!]] = \llcorner z \lrcorner \tag{5.1'}$$

$$e^*[\vec{a} \mapsto \vec{\imath}][x \mapsto u][x_0 \mapsto \mathbf{fst}(u)] \ldots [x_k \mapsto \mathbf{snd}^k(u)] \longrightarrow_{\Theta_1{}^*} \langle v', z'\rangle \tag{5.2'}$$

We can use Lemma 5.15 on page 179; its preconditions are satisfied:

- From the fact that $G[\vec{\imath}](u)$ type-checks under $\Gamma_F, G : \Pi\,\vec{a} : \vec{\gamma}\,.\,\rho_1 \to \rho_2$, we can deduce that the substitution

  $$[\vec{a} \mapsto \vec{\imath}][x \mapsto u][x_0 \mapsto \mathbf{fst}(u)] \ldots [x_k \mapsto \mathbf{snd}^k(u)]$$

  is type-sound with respect to $\vec{a} : \vec{\gamma}$ and $x : \rho_1, x_0 : \rho_1^0, \ldots, x_k : \rho_1^k$.
- The environment $(\Psi[G^c \mapsto \varphi'])$ has the required properties, for $F^c \in \mathbf{dom}(\Theta)$ by assumption, and for G^c by induction hypothesis.

□

Part 2 of Theorem 5.4 on page 163 follows from Lemma 5.16 on the preceding page with a straightforward induction proof over the number of user-defined functions in program p.

Result of extraction is a recurrence

The intuition behind the test described in Section 5.4.5 of whether the extracted cost bound is a recurrence is to use the constraint information contained in $\mathrm{DML}_0^{\Pi}(C)$ type derivations for showing that the argument on each recursive call decreases. Without further proof, however, such an argument only shows the termination of the examined DML program. We further need to argue that conclusions drawn from constraint information about index arguments to recursive function calls also holds for arguments to the corresponding calls occurring within an occurrence equation:

Theorem 5.17 *Assume that the extraction algorithm annotates every call* $F^c\ \vec{\imath}$ *with the index context it was extracted under, i.e.,* $\phi';\Gamma' \vdash F[\vec{\imath}](e'):\rho'$ *gives rise to* $F^{c,\phi'}\vec{\imath}$*, and that the semantics* $\mathcal{T}[\![\cdot]\!]$ *ignores such annotations. If* $\phi;\Gamma \vdash e:\rho \blacktriangleright t$ *and* $\vdash \theta : \phi$*, then, in the unfolded definition of* $\mathcal{T}[\![t]\!]\Psi\theta$*, for all applications* $\Psi(F^{c,\phi'})\ \mathcal{I}[\![\vec{\imath}[\theta']]\!]$ *we have* $\vdash \theta' : \phi'$*.*

Proof: As in the proof of Lemma 5.15 on page 179, we use structural induction over the type derivation of e. The only interesting case is that of a case expression **case** e **of** $p_0 \Rightarrow e_0 \mid \ldots p_k \Rightarrow e_k$. The extracted recurrence-equation term is

$$t + (\mathbf{cond}\ br_0 \mid \ldots \mid br_k)$$

where t is extracted from e by

$$\phi;\Gamma_F,\Gamma \vdash e:\rho' \blacktriangleright t$$

and every br_j from the type-derivation of a branch $(p_j \Rightarrow e_j)$ as

$$\Phi_{j1} \rightarrow \forall(\mathbf{dom}(\phi'_j)\backslash\mathbf{dom}(\theta_j)).\Phi_{j2} \rightarrow t[\theta_j]$$

where

$$\begin{array}{l} p \downarrow \rho' \rhd (\phi'_j;\Gamma'_j) \\ \phi,\phi'_j;\Gamma,\Gamma'_j \vdash e_j:\rho \blacktriangleright t_j \\ \Phi_{j1} = \exists(\mathbf{dom}(\phi,\phi'_j)\backslash\mathbf{var}(\rho')).\mathcal{C}(\phi,\phi'_j) \\ \theta_j = mk\_subst_{\mathbf{dom}(\phi'_j)}(\mathcal{C}(\phi'_j)) \\ \Phi_{j2} = \exists(\mathbf{dom}(\theta_j)).\mathcal{C}(\phi'_j) \end{array}$$

For $\mathcal{T}[\![t]\!]\Psi\theta$ we can simply use the induction hypothesis. For the conditional, however, we have to examine all possible calculations $\mathcal{T}[\![t_j[\theta_j]]\!]\Psi(\theta[\vec{a}\mapsto\vec{z}])$ (with $\vec{a} := \mathbf{dom}(\phi'_j)\backslash\mathbf{dom}(\theta_j)$). With a straightforward structural induction on t_j one can show that

$$\mathcal{T}[\![t_j[\theta_j]]\!]\Psi(\theta[\vec{a}\mapsto\vec{z}]) = \mathcal{T}[\![t_j]\!]\Psi(\theta_j \circ (\theta[\vec{a}\mapsto\vec{z}])).$$

To use the induction hypothesis, it remains to show that $\vdash (\theta_j \circ \theta[\vec{a}\mapsto\vec{z}]) : (\phi,\phi'_j)$, which follows with a short derivation using Lemma 5.13 on page 177 and the assumptions about ϕ, ϕ'_j, θ, θ_j, and $\vec{z}$. □

Theorem 5.17 shows that index-based reasoning about the index arguments to functions in a DML program carry over to the corresponding recurrence equation, hence the third part of Theorem 5.4 on page 163 holds.

Bibliography

[1] Lennart Augustsson. Cayenne—a language with dependent types. In Paul Hudak and Christian Queinnec, editors, *Proceedings of ICFP'98*, pages 239–250, Baltimore, Maryland, September 1998. ACM Press.

[2] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14(2/3), 2001. To appear.

[3] Karl Crary and Stephanie Weirich. Resource bound certification. In Thomas Reps, editor, *Proceedings of POPL'00*, pages 184–198, Boston Massachusetts, January 2000. ACM Press.

[4] Daniel Le Métayer. ACE: An automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, April 1988.

[5] Yanhong Annie Liu and Gustavo Gómez. Automatic accurate time-bound analysis for high-level languages. In Frank Mueller and Azer Bestavros, editors, *Proceedings of the ACM SIGPLAN 1998 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, number 1474 in LNCS, pages 31–40, Montréal, Canada, June 1998. Springer-Verlag.

[6] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[7] Brian Reistad and David K. Gifford. Static dependent costs for estimating program execution time. In Carolyn L. Talcott, editor, *Proceedings of LFP'94*, LISP Pointers, Vol. VII, No. 3, Orlando, Florida, June 94. ACM Press.

[8] Mads Rosendahl. Automatic complexity analysis. In Joseph E. Stoy, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture '89*, pages 144–156, London, September 1989. ACM Press.

[9] David Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, Department of Computing, Imperial College, University of London, September 1990.

[10] Philip Wadler. The essence of functional programming. In Andrew W. Appel, editor, *Proceedings of POPL'92*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.

[11] Hongwei Xi. de Caml. A prototype implementation of DML, based on Caml-light. Available from `http://www.ececs.uc.edu/~hwxi/DML/DML.html`.

[12] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

[13] Hongwei Xi. Dependently typed data structures. In Chris Okasaki, editor, *Proceedings of Workshop of Algorithmic Aspects of Advanced Programming Languages (WAAAPL '99)*, pages 17–32, Paris, September 1999. Available from `http://www.cs.columbia.edu/~cdo/waaapl99.pdf`.

[14] Hongwei Xi. Dependent types for program termination verification. In *Proceedings of 16th IEEE Symposium on Logic in Computer Science*, Boston, Massachusetts, June 2001. IEEE Computer Society Press.

[15] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In Keith D. Cooper, editor, *Proceedings of PLDI'98*, pages 249–257, Montreal, June 1998. ACM Press.

[16] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In Alex Aiken, editor, *Proceedings of POPL'99*, pages 214–227, San Antonio, Texas, January 1999. ACM Press.

Recent BRICS Dissertation Series Publications

DS-01-6 **Bernd Grobauer.** ***Topics in Semantics-based Program Manipulation*****. August 2001. PhD thesis. ii+x+186 pp.**

DS-01-5 **Daniel Damian.** ***On Static and Dynamic Control-Flow Information in Program Analysis and Transformation*****. August 2001. PhD thesis. xii+111 pp.**

DS-01-4 **Morten Rhiger.** ***Higher-Order Program Generation*****. August 2001. PhD thesis. xiv+146 pp.**

DS-01-3 **Thomas S. Hune.** ***Analyzing Real-Time Systems: Theory and Tools*****. March 2001. PhD thesis. xii+265 pp.**

DS-01-2 **Jakob Pagter.** ***Time-Space Trade-Offs*****. March 2001. PhD thesis. xii+83 pp.**

DS-01-1 **Stefan Dziembowski.** ***Multiparty Computations — Information-Theoretically Secure Against an Adaptive Adversary*****. January 2001. PhD thesis. 109 pp.**

DS-00-7 **Marcin Jurdziński.** ***Games for Verification: Algorithmic Issues*****. December 2000. PhD thesis. ii+112 pp.**

DS-00-6 **Jesper G. Henriksen.** ***Logics and Automata for Verification: Expressiveness and Decidability Issues*****. May 2000. PhD thesis. xiv+229 pp.**

DS-00-5 **Rune B. Lyngsø.** ***Computational Biology*****. March 2000. PhD thesis. xii+173 pp.**

DS-00-4 **Christian N. S. Pedersen.** ***Algorithms in Computational Biology*****. March 2000. PhD thesis. xii+210 pp.**

DS-00-3 **Theis Rauhe.** ***Complexity of Data Structures (Unrevised)*****. March 2000. PhD thesis. xii+115 pp.**

DS-00-2 **Anders B. Sandholm.** ***Programming Languages: Design, Analysis, and Semantics*****. February 2000. PhD thesis. xiv+233 pp.**

DS-00-1 **Thomas Troels Hildebrandt.** ***Categorical Models for Concurrency: Independence, Fairness and Dataflow*****. February 2000. PhD thesis. x+141 pp.**

DS-99-1 **Gian Luca Cattani.** ***Presheaf Models for Concurrency (Unrevised)*****. April 1999. PhD thesis. xiv+255 pp.**