C Program Specialization

Master's Thesis
(revised version)

Lars Ole Andersen

DIKU, Department of Computer Science,
University of Copenhagen,
DK-2100 Copenhagen Ø,
Denmark.
E-mail: `lars@diku.dk`

May 1992

Abstract

Automatic program specialization has numerous application areas ranging from specialization of scientific computation to automatic compiler generation. During the last decade, several automatic partial evaluators have been developed and demonstrated their usefulness. However, none of these have both been for a *typed imperative* language and *self-applicable*. The main content of this thesis is the development of an automatic self-applicable program specializer for a substantial subset of the C programming language.

The use of an imperative language introduce many problems not present in partial evaluation of functional languages. New problems addressed and solved in this thesis includes handling of static side-effects under dynamic control, treatment of the imperative data structures arrays and structures, partially static data structures and specialization time splitting of these, and representation of values and programs for efficient self-application.

In the thesis we recall the foundation of partial evaluation and reformulate basic concepts in a typed framework. We define a kernel subset of the C programming language, and specify the semantics formally by the means of an operational semantics.

Based on an analysis of new problems we provide a formally specification of a C program specializer. We define a two-level language, impose well-annotatedness requirements, and finally provide a two-level operational semantics. The two-level semantics specify the program specializer.

A binding time analysis is developed via type inference. The analysis can be implemented efficiently via a constraint set solving algorithm. The same technique is applied in the construction of an untagging analysis, which is used for simplification of the types in residual programs.

In the last part of the thesis some experimental results are reported and assessments given.

To the best of our knowledge this is the first successfully developed and implemented self-applicable program specializer for a "real-scale", imperative and typed language.

Preface

This report is a revised version of the author's Master's Thesis ("speciale") submitted in partial fulfillment of the requirements for a Danish Cand. Scient. degree ("kandidatgrad") [Andersen 1991a]. It contains work done in the period March 1991 to December 1991 under the supervision of Prof. Neil D. Jones at DIKU, Department of Computer Science, Copenhagen University. The revision was done in May, 1992. This work has partly been published in [Andersen 1992b] and [Andersen 1992a].

All work reported in this thesis has been made solely by the author. As always, though, parts of the work has been based on work by others. We have tried painstakingly to provide references and clearly state the origin of the applied techniques. In the conclusion an extensive comparison with related work can be found.

The thesis consist of eight chapters where the first is an introduction and the last a conclusion. The purpose of the introduction is to provide the background for this work, and motivate its usefulness. We also give a brief overview of the basic concepts necessary for full benefit of the report. The last section of the introduction contains an overview of the rest of the thesis. The conclusion closes the thesis with a discussion of related work, several directions for future work, and finally—the conclusions! The most of the material in this technical report is a revised version of [Andersen 1991a]. Chapter 3 has been extended considerably, Chapter 7 almost rewritten, and Chapter 8 extended with new material. Several of the programs we have implemented and generated automatically can be found as appendices.

The work in this thesis is both theoretical and practical. We report a semantically based development of an automatic partial evaluator for a subset of C, beginning with an operational semantics for the treated C subset, to a problem analysis, to a complete formal specification of the C program specializer. We believe that even though we focus on the C language here, many of the techniques are applicable in other contexts aswell. The implementation we have done illustrate the theory and demonstrates that it works!

For obtaining full benefit of this thesis some prerequisites are necessary. The subject of the thesis is *program specialization*, and we expect knowledge corresponding to the references [Jones *et al.* 1989] and [Gomard and Jones 1991a]. Part of the material will be reformulated in our setting, but the general foundation will not be repeated here. Furthermore, knowledge of the syntax and semantics of the C programming language [Kernighan and Ritchie 1988] is required, and we expect some practical experience too. For specification of operational semantics we use a style resembling *Structural Operational Semantics* [Plotkin 1981] and Natural Semantics [Kahn 1987]. The general framework of types and typing is used without detailed explanation [Milner 1978].

Acknowledgements

Neil D. Jones has been the supervisor of this project. I would like to express my thanks for his encouraging enthusiasm and valuable discussions. The amazing ability of knowing what the central problems are has been extremely useful, especially in the early days of this project.

Special thanks to Peter Sestoft for his extensive and thorough comments on drafts of this thesis. These have for sure improved the presentation of the material with orders of magnitude, and came then they were really needed. Also thanks to Carsten Gomard for useful discussions during and after the project.

Fritz Henglein proposed the use of contraint set based type inference for binding time analysis. I would like to thank him for his comments on the analysis, and for clarifying some subtle points.

Warm thanks goes to the TOPPS/Sematique group at DIKU for providing an always lively and fruitful environment. Its member are always helpful and ready to share their knowledge—and to have fun! Without the work done by the researches in the TOPPS group, this work would probadly not have been.

Thanks to the people who have shown their interest during and after the project. Beside the people mentioned above, among others Anders Bondorf, Robert Glück and Torben Mogensen.

Contents

Chapter 1

Introduction

A large class of similar problems can essentially be solved in two ways: one can write a specific program for each problem, or one can write a general, parameterized program solving all the problems. In practise, the former solution is often preferred due to one urgent reason: *efficiency*. A program tailored to a given specific problem is nearly always faster than a general version which has to test for various cases of the problem specifications. The general problem solver possesses, however, several fruitful characteristics. It is faster to program, it is frequently more manageable to maintain due to its generality, and last but not least: a doubling of the number of problems to solve does not imply that the poor programmer has to work overtime!

Is it then possible to get the best of the two worlds: *generality* and *efficiency*? Yes! By specialization of the general program to a given problem specification, an efficient program tuned to a specific problem instance can be obtained, but still, only one program has to be written by hand. Can the program specialization be done automatically on the computer? Yes! One of the main purposes of this thesis is to demonstrate this in a more applied context than earlier work: a subset of the C programming language.

1.1 Background

During the last decade, *partial evaluation*, or *program specialization*, has proven its usefulness in numerous areas. The applications range from sophisticated program optimizations, for example specialization of scientific computation [Berlin and Weise 1990] and ray-tracing [Mogensen 1986], and programming language implementation, for example compiler generation [Bondorf 1990a] and proto-type implementation [Heldal 1991]. In common for these applications are that a one-stage program is converted into a two stage version. For instance, an *interpreter* is a one-stage program taking as input both a program and some data. A compiler, on the other hand, is a two stage program: the compiler takes as input a program and generates a new program which is subsequently applied to the input data. It is well-known that the latter approach usually is more efficient than the former. When running the compiled program, all the book-keeping in the interpreter is not present. This is often referred to as the *interpretation overhead*.

The success of program specialization can be ascribed to several things, but two im-

portant points are *automation* and *correctness*.

Automation is evidently important. If the user has to guide the generation of the specialized program, errors are likely to creep in. This would devalue the use of a partial evaluator as a tools. Furthermore, when generation of stand-alone compilers is attempted, many complex run-time configurations from the partial evaluator arises—a program the user normally have no knowledge about. As illustrated in [Pagan 1990], to convert even small programs into "specializing form" may require a considerably human effort. From a pragmatically point of view, automation is irrefutable.

Correctness is, of course, indisputable. A program generated automatically by a program generator must be faithful to the specification from which it was derived. Hence, a partial evaluator must be *semantic preserving*. Analogies: a compiler must obey the semantic rules of the subject language, and a Yacc produced parser must accept the language specified by the input grammar.

For satisfying both these criteria, the techniques applied can only be of a certain complexity. The basic techinques in program specialization are *specialization* of program points[1] to known values, *reduction* of partially known expressions, *evaluation* of completely known expressions, and *transition*[2] *compression*. In practise, these simple methods give surprisingly good results: many programs can be speeded up by a factor 10 which is often sufficient for the application. On the other hand, no more than *linear* speedup can be obtained this way. In [Andersen and Gomard 1992] it is proven that *superlinear* speedup is impossible. Superlinear speedup *can* arise through *e.g.* introduction of sharing, or dicarding of computation, but such transformations are in general unsafe.

The theoretical foundation of partial evaluation is Kleene's S-m-n theorem known from recursion theory. Partial evaluation can be seen as the *efficient* implementation of the proof [Jones 1990]. The research in automatic program specialization can roughly be divided into two groups: self-applicable partial evaluation and non-self-applicable partial evaluation. Experiments in the latter began in the mid-seventies in a Lisp setting [Haraldsson 1978], and current research now both consider functional, logic and imperative languages [Weise *et al.* 1991], [Meyer 1991].

When generation of stand-alone compilers is desired, the partial evaluator must be self-applicable. In general, if the partial evaluator is self-applicable, a *program generator* can be made. As input it takes a general program and parts of its input data, and produces a specialized version. The first succesfully developed, implemented and self-applied partial evaluator was made by the Copenhagen Mix project [Jones *et al.* 1989] which treated a small Lisp-like language. Later the technologies were ported to stronger languages, *e.g.* a higher-order subset of the Scheme language with some imperative effects [Bondorf 1990a]. A few experiments with *typed* languages have been execited [Launchbury 1991], and simple untyped imperative flow-chart languages using S-expressions as the sole data structure [Gomard and Jones 1991a]. Still missing is a self-applicable partial evaluator for a large-scale typed imperative language with imperative mutable data structures.

There are good justifications for a self-applicable partial evaluator for an imperative language, though. Imperative programs usually run more efficiently on the computer

[1] Functions and labelled statements

[2] Function calls or jumps

than their functional counterparts; many applications are most naturally expressed in imperative terms, *e.g.* sorting algorithms; the lack of side-effects in functional languages often causes enormous memory usage due to (unnecessary) copying, and last but not least: imperative languages are in wide spread use and are totally dominating the programming language market. Consider for instance areas such as scientific computation in physics and data base systems in the industrial. Furthermore, it is general believed that types are *good*. They catch many errors at compile-time and may help the programmer to clarify her mind during the program development. Treatment of types in connection with self-applicable partial evaluation has previously been deferred due to representation problems, but this is obviously problems which must be overcome if partial evaluation is to evolve into a *programming tools*. For example, in the "real" world, there is no good prospective market for compilers from (efficiently implementable) Algol 68 to untyped functional Scheme [Consel and Danvy 1991]!

In this thesis we report the development of a self-applicable program specializer for a substantial subset of the C programming language [Kernighan and Ritchie 1988]. The subset includes global variables, and functions with both parameters and local variables. The treated data structures are base type variables, structures, multi-dimensional arrays, and a restricted use of pointers. Almost all ANSI C statements and expressions are allowed. Not treated by the methods described here is general heap-allocated data structures and pointers. We discuss and outline possible ways to handle these in the end of the thesis. We have implemented a proto-type version of the C program specializer and applied it in a number of experiments, including some of the projects studied in [Pagan 1990].

For archieving efficient self-application, *binding time analysis* is now a well-established tool. The aim of binding time analysis is to compute which statements (expressions) that can be performed at specialization-time, and hereby provide the specializer with a maximum of *compile-time* information. Originally, binding time analyses were carried out by abstract interpretation over the simple binding time domain $S \sqsubset D$, where S denotes "specialization-time" and D "run-time" [Jones *et al.* 1989]. More refined analyses taking care of higher order languages and *partially static structures* have later been developed [Bondorf 1990a], [Mogensen 1988]. It is, however, well-known that abstract interpretation has poor worst-case complexity, and is often intractable on even small programs. In this work we take another approach and base the binding time analysis on *type inference* using *constraint solving*. This has been done for the untyped lambda calculus [Gomard 1990] and an algorithm with almost linear run-time has been developed [Henglein 1991b]. The use of an imperative language, and the semantics of C introduces several new aspects, though.

The use of a typed language has a significant impack on the programs generated by self-application. The types in the programs are too general, and all values are tagged. In order to improve the efficiency of the residual programs, we introduce an *untagging analysis* which aims at *simplifying* the types when it is safe to do so. The analysis is based on type inference and can be implemented using the same techniques as the binding time analysis. We believe this illustrates the generality of program analyses based on type inference and constraint set solutions wery well, and find the subject interesting in itself.

1.2 This Thesis

The main goal of this work is development of an automatic self-applicable partial evaluator for a substantial subset of the C programminng language [Kernighan and Ritchie 1988]. Although this work is based around the C programming language, we believe several of the results are applicable in both partial evaluation of other imperative languages, *e.g.* Pascal, and of languages with states, *e.g.* object oriented languages. Furthermore, we address problems with self-application and types, which are independent of the particular language in use.

We identify the main goals addressed in this thesis as the following:

- A kernel part of C is identified and defined via an operational semantics.
- Self-representation of values for self-application is considered.
- Development of a self-applicable partial evaluator for the kernel language.
- Development of an automatic binding time analysis.
- Development of an untagging analysis.
- Results and assessments from a proto-type implementation are reported.

The basic underlying principles are *automation* and *correctness*.

To guarantee semantic correctness of the partial evaluator, a well-defined language is needed. This is particular important in the case of C, which possess a very non-restrictive semantics. We derive a kernel language and define it formally via an operational semantics. The semantics is then extended to a *two-level semantics* which captures program specialization. The semantics of the kernel language is consistent with the ANSI C definition, but do clarify some minor points.

A partial evaluator operates on programs and values as data objects. Since it is supposed to work on all type of values, a uniform data representation is required. In untyped languages like Scheme, the input reader and the dynamic typing converts all values into a uniform representation, but in C, the encoding must be performed explicitly. The data representation is complicated by the possibility of user defined abstract data structures, *e.g.* structures, which may have an arbitrary size. This involves the problem of *double encoding* that may prevent efficent self-application. Furthermore, the parameter passing semantics must be preserved by the representation, for example, arrays are passed call-by-reference, but structures call-by-value.

The call-by-reference of arrays can introduce aliaes and sharing. This impose requirements upon the binding time separation, and complicates the handling of the specialization time store. During the specialization, several copies of the specialization time store may exist, and it must be assured that *e.g.* pointers are correctly initialized. The possibility of side-effects on non-local variables complicate function specialization. This is mainly due to the lack of referential transparency. Furthermore, side-effects on non-local variables may be under dynamic control which render the specialization of statements after a function call difficult. These problems are not present in purely functional languages.

In order to obtain an *optimal* specializer, the representation of the values in the specializer is important. If an array is used, it must be split into separate variables, but the semantics of C do not always allow an array to be split. In general we consider *partially static data structures* and analyze when it is safe to split these during the specialization.

The C program specializer is formally specified via a two-level operational semantics. In the two-level semantics, *well-annotatedness* plays the same role as types in the standard semantics: it prevent the specializer from "going wrong". The presence of pointers and non-local side-effects introduce constraints between different parts of the program which must be satisfied. We specify well-annoatedness in form of type inference systems that leads to a binding time analysis in a natural way.

Traditionally, binding time analysis has been based on abstract interpretation. In this work we develop a binding time analysis based on type inference. The analysis consists of four parts. First, a *constraint set* is collected, representing binding time properties of the program. Then the constraint set is *normalized* in a solution preserving way, a solution to a normalized constraint set can then easily be found, and an annotation derived. It can be shown that the analysis can be implemented to run in almost linear time using find/union data structures.

The use of a typed language has an impack on the types in residual programs. Assume a self-interpreter is specialized to a program p yielding a residual program p'. The self-interpreter must as the specializer represent all values in a single union data type, as it must be able to accept all types of input. However, the types in p' are inherited from the self-interpreter, and will hence all be of this union type. This is inefficient and impede optimality. To simplify the types, we have developed an *untagging* analysis based on type inference. The analysis detects when it is *safe* to replace a union type variable with *e.g.* an `int` variable.

We have made a proto-type implementation of parts of the partial evaluator and applied in a number of experiments. We report some results and provide benchmarks. It is shown that a complete layer of interpretation can be specialized away, and compiler generation by self-application is demonstrated. Several of the examples given stems from [Pagan 1990] and hence illustrates that automation is possible.

1.3 Program Specialization and Compiler Generation

A *partial evaluator*, or *program specializer*, is for historical reasons often named *mix*. A main characteristic of a partial evaluators is whether it is *on-line* or *off-line*.

In *on-line* partial evaluators, the separation of *specialization-time*[3] (*i.e.* what can be done during specialization) and *run-time* (*i.e.* what must be deferred to run-time) is done a *during* the specialization. The partial evaluators [Weise *et al.* 1991] and [Meyer 1991] are both on-line. In an *off-line* partial evaluator, the separation is performed *a priori* to the actually specialization by the means of a binding time analysis. The systems

[3]Specialization-time is often also referred to as compile-time

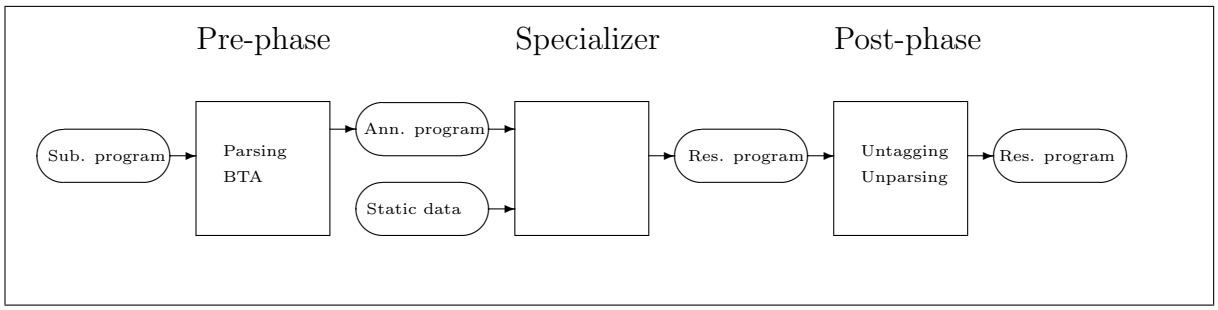

Figure 1: The structure of an off-line partial evaluator

[Bondorf 1990a], [Gomard and Jones 1991a], [Gomard and Jones 1991b] are all off-line and self-applicable.

It has been argued that on-line specialization of imperative languages is more appropiate than off-line, as in on-line systems the binding time state of variables may change during the processing. However, then the goal is *self-application* experience clearly demonstrates that off-line techniques are superior [Bondorf *et al.* 1988]. All succesfully implemented and self-applied partial evaluators have been off-line, and the use of binding time analysis for providing the specializer with a maximum of compile-time information is essential. In this thesis we will solely consider off-line partial evaluation.

When considered as a system, an off-line *mix* usually consists of three parts: a pre-phase, the program specializer and a post-phase. This is illustrated in Figure 1.

The pre-phase includes parsing, and most foremost: binding time analysis. Other analyses goes here aswell, *e.g.* livevariable analysis and call-graph computation. The input to the pre-phase is the *subject program*; output is a binding time annotated subject program. The specializer specializes the annotated program to the given *static* values. Hereby a *residual program* is produced, which is handed over to the post-phase. The post-phase consists of various analyses, *e.g.* untagging analysis, and unparsing. The final result is a new executable (or compiler-ready) program accepting the *dynamic input*.

In this introductory section we will use the terms mix and specializer interchangeably in agreement with the literature. In later chapters we will explicitly make a distinction between mix as a system and the specializer as a part of mix.

EXAMPLE 1.1 Consider the for partial evaluation canonical program *power* computing the power function. In can be programmed in ANSI C as follows.

```
/*  Compute x to the n'th */
int power(int n, int x)
{
   int pow;
   pow = 1;
   for (; n; n--)
      pow *= x;
   return pow;
}
```

Assume that `n` is statically given the value 3, but that `x` is unknown. Then, the computation involving `n` can be performed at specialization-time (`n` decremented and the loop unrolled), but the actions depending upon `x` must be deferred. This is determined by the binding time analysis. A residual program for *power* is:

```
int power_3(int x)
{
   int pow;
   pow = 1;
   pow *= x;
   pow *= x;
   pow *= x;
   return pow;
}
```

The `power_3` function is a specialized version of `power()` computing `x` to the 3rd. The specializer developed in this thesis will produce the above program when applied to *power* and 3. End of Example

Program Run A specializer is a program taking as input a(nother) program and some of its input data. It is therefore important to distinguish between program texts, representations of programs and the meaning of programs.

Let $\mathcal{C}$ be a programming language over the data domain $\mathcal{D}^{\mathcal{C}}$. Elements in $\mathcal{C}$ are program texts which may or may not satisfy syntactical and semantical requirements imposed upon the language. Concretely, in this report $\mathcal{C}$ will usually be the C programming language and $\mathcal{D}^{\mathcal{C}}$ integers, characters, multi-dimensional arrays *etc.* We will often omit superscripts when the language in use is clear from the context. Without loss of generality we take the simplified view that a program is a function from some input data to a result value. Hence no input of data through reading of files, or the like, is possible. We adapt the convention that if example programs possesses no `main()` function, the first function is taken to be the initically called function.

The net effect of running a program on a $\mathcal{C}$ computer can thus be described as a function as follows.

Notational Convention 1.1
Let $\mathcal{C}$ be a programming language over data domain $\mathcal{D}^{\mathcal{C}}$ taking n input values. The net effect of applying p to the values $d_1, \ldots, d_n \in \mathcal{D}^{\mathcal{C}}$ on a "$\mathcal{C}$" machine is denoted:

$$[\![p]\!]_{\mathcal{C}}\,(d_1,\ \ldots,\ d_n) \Rightarrow d, \tag{1.1}$$

where $d \in \mathcal{D}$ is the result. Non-termination or error is represented by $\perp$. The result value will be omitted when convenient.

Observe the differences: p is a program text and $[\![p]\!]_{\mathcal{C}}$ is a function from $\mathcal{D}^n$ to $\mathcal{D}$, that is, its meaning.

EXAMPLE 1.2 Consider the *power* program from before. Take the `power()` function to be the main function of the program. Computing 2 to the 3rd is thus written

$$[\![power]\!]_C\,(2,3) \Rightarrow 8$$

yielding result 8, and it holds that $[\![power]\!]$ equals the mathematical power function for all non-negative `n`. END OF EXAMPLE

Interpreters and program representation An *interpreter* or *evaluator* for a language $\mathcal{C}'$ (written in language $\mathcal{C}_{int}$) is a program accepting as input a program p (in $\mathcal{C}'$) and its data $d_1, \ldots, d_n$, and yields as output the value d as normal execution of p would.

In languages like Scheme, a program is a legal data object and can be given directly to a self-interpreter. In C, a suitable program representation must be devised, for example, a program can be representated by an array (of functions) of arrays (of statements). The lack of types[4] makes the representation of values straightforward. In typed languages, the representation problem is more prevalent.

Since an interpreter is supposed to work on all programs taking different types of input, the input data d_i must be embedded into a *single uniform* data type `Val`. This is done automatically in languages like Scheme (by the `read` function), but in C, the conversion must be explicit via encoding functions. For simplicity we will for now ignore that a program only can take a fixed number of inputs. Further, as we are aiming towards self-application we take the language of the interpreter and the interpreted language to equal.

NOTATIONAL CONVENTION 1.2
Let $d \in \mathcal{D}$ be a data value of some type. The encoding of d into an (unspecified) data structure of type t is denoted by $\overline{d}^t$. We overload the notation to programs, e.g., $\overline{p}^{Pgm}$ denotes the representation of program p in a `Pgm` *data structure.*

In C, the union type `Val` could be a structure with a huge `union` inside.

EXAMPLE 1.3 Let *int* be an C interpreter. Running the *power* program on the integers 2 and 3 correspond to

$$[\![int]\!]_C\left(\overline{\overline{power}^{Pgm}}, \overline{\overline{2}^{Val}}, \overline{\overline{3}^{Val}}\right) \Rightarrow\ \overline{\overline{8}^{Val}}$$

where programs are represented in a data structure of type `Pgm` and values are encoded into a union type `Val`. The result is the encoding of the number 8 in the `Val` type. To get the number "8", the integer part of `Val` has to be projected out. END OF EXAMPLE

Assume that we apply the self-interpreter to itself, that is, informally:

$$[\![SelfInt]\!]_C\,(SelfInt, p, d)$$

[4] In fact, there is one type!

where p is a program and d some data.

The interpreted self-interpreter must of course be represented in a `Pgm` data structure, but what about the program p and the input data d? As input to the self-interpreter, it must be represented as a `Pgm` data structure, but as a value input to the running version of *SelfInt*, it must subsequently be encoded into the `Val` type. Similar for the data d. In other symbols:

$$[\![\mathit{SelfInt}]\!]_C \left(\overline{\mathit{SelfInt}}^{Pgm}, \overline{\overline{p}^{Pgm}}^{Val}, \overline{\overline{d}^{Val}}^{Val} \right).$$

The same observation was made by Launchbury when he designed a partial evaluator for a subset of LML. Our notation is similar to his but more informative [Launchbury 1990].

Program specialization Normally, it make only sense to apply an interpreter to a program p and *all* its input data. A partial evaluator is a "smart" evalautor which can perform such partial evaluations. The output from a partial evaluator is not a value, but a *residual program* specialized to the given values. The residual program computes, when applied to the rest of the input data, the same values as p would if applied to all its input data. This characteristic is captured in the *Mix-Equation*[5]. Since *mix* essentially is an evaluator, the same encoding requirements applies for *mix* as for an ordinary evaluator.

Definition 1.1 (The Mix Equation)
Let p be a program taking input data $d_1, \ldots, d_n \in \mathcal{D}$. A partial evaluator mix is a program (in $\mathcal{C}$) such that if

$$[\![p]\!]_{\mathcal{C}} (d_1, \ldots, d_n) \Rightarrow d.$$

then

$$[\![\mathit{mix}]\!]_{\mathcal{C}} \left(\overline{p}^{Pgm}\ \overline{d_1}^{Val}, \ldots, \overline{d_i}^{Val} \right) \Rightarrow \overline{p'}^{Pgm} \quad \textit{and} \quad [\![p']\!]_{\mathcal{C}} (d_{i+1}, \ldots, d_n) \Rightarrow d$$

provided mix terminates. □

We have taken the language of the subject program p and of the partial evaluator to equal. This is necessary if *mix* is to be self-applied. The Mix Equation generalize easily to an arbitrary permutation of the input data d_i. For a thorough treatment of the Mix-Equation we refer to [Jones *et al.* 1989].

The part of the input given to *mix* is called the *static input*. The rest is called the *dynamic* input. The *mix* produced program is called a *residual* program, and is a specialized version of the subject program with respect to the given static values. A more appropiate word for partial evaluator is hence *program specializer*.

The possibility of stand-alone compiler generation [Futamura 1971] and compiler generator generation [Beckman *et al.* 1976] are now known as the *Futamura* projections. We restate these in a typed version where encoding into the union type is explicit.

[5] the *mix-equation* is an instance of Kleene's S-m-n theorem

THEOREM 1.1 (THE FUTAMURA PROJECTIONS)
Let $\mathcal{C}$ be a programming language over data domain $\mathcal{D}$, and int an interpreter in $\mathcal{C}$ (interpreting an unspecified language). Let p be a program in the language interpreted by int.

1. *Futamura:* $[\![mix]\!]_{\mathcal{C}} \left(\overline{int}^{Pgm},\ \overline{p}^{Val}\right) \Rightarrow \overline{target}^{Pgm}$
2. *Futamura:* $[\![mix]\!]_{\mathcal{C}} \left(\overline{mix}^{Pgm}\ \overline{\overline{int}^{Pgm}}^{Val}\right) \Rightarrow \overline{compiler}^{Pgm}$
3. *Futamura:* $[\![mix]\!]_{\mathcal{C}} \left(\overline{mix}^{Pgm},\ \overline{\overline{mix}^{Pgm}}^{Val}\right) \Rightarrow \overline{cogen}^{Pgm}$

provided mix terminates. □

The first Futamura projection states that *compilation* can be done by specialization of an interpreter to a program. The target code will be in the *mix* language. The second projection shows that a *stand-alone compiler* can be generated by *self-application* of *mix* (precisely: the specializer). The compiler compiles programs from the interpreted language to the *mix* language. The third Futamura projection describes *generation of a compiler generator*. Given an interpreter, *cogen* generates a stand-alone compiler.

The verification of the projections is straightforward and thus omitted. Proofs can be found in *e.g.* [Jones 1988]. Observe the double encoding of the value arguments when *mix* is self-applied. We address in detail the interaction between double encoding and self-application in the next chapter.

Program specialization by two-level interpretation Partial evaluation can be regarded as a non-standard execution of the program. Assume the binding time annotation is given through the syntax, *e.g.* a static `if` and a dynamic `if`. Such a program is often called a *two-level program* [Nielson and Nielson 1988a]. The program specialization is then the semantics of two-level programs $2\mathcal{C}$. With this interpretation in mind, the Mix Equation can be reformulated as follows.

DEFINITION 1.2 (THE MIX EQUATION)
Let p be a program in $\mathcal{C}$ and p^{ann} a binding time separated program, i.e. a two-level program in $2\mathcal{C}$. Let d_1 and d_2 be input data to p. Then holds that:

$$[\![p^{ann}]\!]_{2\mathcal{C}} d_1 \Rightarrow p' \quad \textit{and} \quad [\![p']\!]_{\mathcal{C}} d_2 \Rightarrow d \quad \textit{iff} \quad [\![p]\!]_{\mathcal{C}}(d_1, d_2) \Rightarrow d.$$

The residual program p' is generated by the execution of the two-level program p^{ann} under the semantics $2\mathcal{C}$. □

This is the view of program specialization which will be exploited in this thesis. We will not, however, be completely precise with termination properties but tacitly assume that the specializer terminates except when otherwise stated.

1.4 The Treated Subset of C

We believe the following chapters provide a fairly good description of what precisely can be handled by the program specializer developed in this thesis, and hence we will not give a formal specification. In this section we give a broad overview of the subset of C treated.

We consider a program as consisting of a collection of modules, *e.g.* a main module for opening and reading of file; an I/O module and several computation modules. The partial evaluator is typical applied to the computation modules, and thus we assume all input is through input parameters. External variables are not allowed.

A program must consist of an optional number of global variable declarations followed by at least one function definition. It is not possible to restrict the scope of global variables via the lexical ordering of functions and declarations.

The supported types includes all the base types (`int`, `char`, `double` ...), structures (but not unions), multi-dimensional arrays and a restricted use of pointers. Heap allocated arrays are allowed, but general heap allocated structures are not treated. Only pointers to arrays are legal, hence it is illegal to apply the address operator `&` to both base type variables and structures. In Section 8.2 we discuss extensions to general heap allocated data structures with possible cyclic pointers and sharing.

A function can declare both parameters and local parameters. Both base type variables, structures and arrays can be passed as actual parameters. All the usually statements can be used, *e.g.* `if`, `while`, `goto`, `return`. Assignments and functions calls are supported as normal. External functions may be called, but they are not allowed to refer to any variables but the parameters and they may not side-effect through call-by-reference parameters (*e.g.* an array pointer). All the ANSI C expression kinds except pre- and post increment/decrement are handled.

The most of the restrictions imposed are syntactically recognizable but a few are on the semantic level. We return to this in the next chapter.

1.5 Overview

The thesis is organized as follows. The present chapter contains an introduction to the goal of this work, and an overview over program specialization especially with the view of typed languages.

Chapter 2 is a formal description of the kernel language Core C used in the program specializer. We define its syntax and semantics, and discuss self-representation of data structures. In the last part of the chapter, a self-interpreter for the language is outlined.

The Chapters 3 and 4 are the main chapters of this report. In these the program specializer for Core C is developed and described. Chapter 3 is a general problem analysis, where we stress new problems mainly due to the language. Chapter 4, on the other hand, is a formal specification of the program specializer. We introduce a *two-level language*, define *well-annotatedness* rules, and provide a complete *two-level semantics*. An implementation of the two-level semantics is the program specializer.

Chapter 5 is devoted to *binding time analysis*. We describe a fully automatic binding time analysis for our language based on the solution of a *constraint set*. The analysis

consists of four parts: *construction* of a constraint set, *normalization* of the constraint set, *finding* a minimal solution, and *annotation* of the program. The analysis presented is an adaption of a similar analysis by Henglein [Henglein 1991b], but with some extensions.

Chapter 6 reports the development of an *untagging* analysis. When a specializer in a typed language is self-applied, the generated residual program has too general types. The purpose of the untagging analysis is to *simplify* the types in the residual program by removal of tagging/untagging operators.

The developed program specializer has been implemented and used in various experiments. In Chapter 7 we report some of the results we have obtained, including an example of compiler generation by self-application. We also briefly give an overview of the implementation, which is rather involved. This is first and foremost due to the destructively modifiable data structures with pointers which demands a complex memory management.

The last chapter contains an extensive comparison with related work with a large number of references to the literature. We also give several directions for future work, and outline some of the possible areas where the work of this thesis can be applied. Finally, a conclusion on the whole work is given.

In the appendices listings of some of the program we have implemented, and some of the automatically generated programs can be found.

Chapter 2

The Core C Language

A subject language for program specialization should ideally meet two contrary goals. From the pragmatic programmer's viewpoint a rich language with many powerful constructs is desirable. This ease programming and improves the usability of the system. On the other hand, the bare core of a program specializer is essentially a self-interpreter, so a reasonably sized language is definitely advisable if self-application is to be achieved.

In this chapter we present and formally describe Core C which is a kernel part of the C programming language [Kernighan and Ritchie 1988]. Included in Core C is global variables, functions with both parameters and local variables, data structures such as base type variables, structures, multi-dimensional arrays and some pointers. There are state-update and control-flow statements, and an essential collection of expression kinds. The Core C language can be considered as a *control-flow* graph representation of C programs. Jumps are via labels, and there is no block structure except for functions. It can easily be seen that Core C captures almost the full ANSI C language, and that the transformation between the two can be performed automatically by means of simple program transformations.

The net effect is that the system can treat almost the full language, that is, the programmer can use all the usual C statements in her programs, but the system needs only to treat the kernel part: Core C. This also implies that *e.g.* the specializer can be written in C, but it only has to interpret Core C constructs. In this thesis we will normally use Core C in examples for illustration purposes, but freely write C programs when no confusion is likely to occur.

C is a *strongly typed* language[1] whereby it differ from the languages previously applied in partial evaluation. For instance, the partial evaluators [Bondorf 1990a], [Consel 1988], [Gomard and Jones 1991a], [Gomard and Jones 1991b], [Weise *et al.* 1991] all consider untyped languages. When types are present, problems with double encoding during self-application arises, as discussed in the previous chapter. The double encoding causes an enormous memory usage and inefficient indirect representation. We show how these problems can be overcome by extending the set of base types in Core C.

The chapter is organized as follows. First, in Section 2.1, we present the syntax and static semantics of Core C. Most of the static semantics is inherited from C, but some extra

[1] A least the part of C we treat!

restrictions are imposed. The *dynamic* semantics is defined via an *operational* semantics in a style resembling Structural Operational Semantics [Plotkin 1981].

In Section 2.2 we discuss data representation and encoding of programs and values. We show how to reduce the problem of double encoding, and outline a self-representation of values in Core C.

The final section contains a short description of a self-interpreter for Core C, with emphasis on representation of arrays and addressing of these. As a program specializer mimics a self-interpreter, a self-interpreter with a good separation of binding times is a prerequisite for efficient self-application.

2.1 The Syntax and Semantics of Core C

Partial evaluation is basically the specialization of *program points* to *static* values. In a functional language the program points are determined by the functions and hence the basic technique in that setting is *function specialization*. In imperative languages targets for jump (`if`, `goto`) are also program points.

It is thus advantageous to have a general control-flow. For example, it is better to express a loop via an `if-goto` construction than a `while`, as in the former case the components may be used in (new) residual loops or specialized versions of it. In the latter case, all the specializer can do with the `while` statement is either to completely eliminate it during specialization, or to suspend it (that is, let it go to the residual program).

In Core C the control-flow is solely via (un)conditional jumps to labels and function calls. Where are no constructs such as `while` and `do-while`. The store modifying commands includes assignments and function calls. In contrast to ordinary C, assignments and calls are at the statement level and not the expression level. Gained hereby is that evaluation of an expression cannot cause side-effects to occur. The "lifting" of assignments and calls to the statement level can be done automatically and represent hence no inconvenience for the user.

The transformation between (almost) full ANSI C and Core C should be obvious, and we therefore omit a formal specification.

2.1.1 Syntax of Core C

For notational convenience, the abstract syntax of Core C resembles the concrete syntax of C. All statement and expression kinds are, however, equipped with a tag for reference purposes. For simplicity we here only consider the basetype `int` as a representative for all basetypes, *e.g.* `long`, `char`, `float`, `double`, ...

Definition 2.1 (Syntax of Core C)
The abstract syntax of Core C is defined by the BNF grammar in Figure 2. Start nonterminal is CC. □

The following notation is adapted. Tokens are written in typewriter face (`int`), classes of tokens are written italic face (*id*), and single literals are quoted (';'). Zero or more

id	$\in$	Variable identifier	
fid	$\in$	Function identifier	
eid	$\in$	External function identifier	
label	$\in$	Label	
const	$\in$	Int	
uop	$\in$	C unary operator	
bop	$\in$	C binary operator	
CC	::=	decl* fundef$^+$	*Core C program*
decl	::=	type *id* typespec	*Variable declarations*
type	::=	`int` | type '$*$' | `struct` *id* '{' decl$^+$ '}'	
typespec	::=	ϵ | typespec '[' *const* ']'	
fundef	::=	type *fid* '(' decl* ')' '{' decl* stmt$^+$ '}'	*Function definitions*
stmt	::=	*label* ':' `assign` lval '=' exp	*Statements*
	|	*label* ':' `return` exp	
	|	*label* ':' `goto` *label*	
	|	*label* ':' `if` '(' exp ')' *label label*	
	|	*label* ':' `call` *id* '=' *fid* '(' exp* ')'	
exp	::=	`cst` *const*	*Expressions*
	|	`var` *id*	
	|	`index` exp '[' exp ']'	
	|	`struct` exp '.' *id*	
	|	`addr` '&' exp	
	|	`uop` *uop* exp	
	|	`bop` exp *bop* exp	
	|	`ecall` *eid* '(' exp* ')'	
lval	::=	`var` *id* | `index` lval '[' exp ']'	*Left-expressions*

Figure 2: Abstract syntax of Core C

repetitions of a non-terminal are denoted by a superscripted star*, one or more by a superscripted plus$^+$.

A Core C program consists of an optional number of global variable declarations followed by at least one function definition. It is not possible to mix up variable declarations and function definitions.

A variable can either be of base type or a compound type. The possible compound types are structures, multi-dimensional arrays and pointers (to arrays). The use of pointers is addressed below.

A function definition consists of declarations of parameters and local variables, and a sequence of labelled statements. A statement can be an assignment (`assign`), an uncondi-

tional jump (`goto`), a conditional jump (`if`), a return from function invocation (`return`), or a function call (`call`). The intuitive meaning of the call statement is as follows: first the actual parameters are evaluated, then the function is called and the result stored in the variable at the lefthand side of the call. It is part of the static semantics that all (Core C) function must return a value, see below. Only base type variables can be assigned this way. This can be overcome by use of temporary variables introduced during transformation from C to Core C.

An expression can be a constant, a variable reference, an array or structure indexing, an application of a unary, binary or external operator/function, or the address operator `&`. The address operator has been separated from the unary operators for notational purposes.

A lefthand side expression can be a variable or an array indexing. Assignment of structure fields can be achieved by a suitable use of external functions using the fact that structures are passed call-by-value. Note, we allow the C indirection `*p`, where `p` is a pointer, but transform it into the equivalent `p[0]` in Core C.

Pre/post increment/decrement is not supported. Thus, (as will become apparent below,) evaluation of an expression is side-effect free.

EXAMPLE 2.1 For dynamic, heap allocation of arrays, the external `calloc` replacement `cmix_alloc_array(int)` is introduced. It returns a pointer of type `Val` (to be defined below) to an array of the appropiate size. For example, the following (C) program fragment allocates an array of lenght 10:

```
Val *p;                          /*  Declare Val pointer  */
p = cmix_alloc_array(10);         /*  Allocate the array     */
p[2] = cmix_int2Val(87);
```

Allocation using `malloc()` is not advisable as it will confuse the memory management in the specializer, see later. END OF EXAMPLE

2.1.2 Static Semantics for Core C

The static semantics of Core C is inherit from C and will thus not be recapitulated here. Instead we refer to the definition [Kernighan and Ritchie 1988]. Recall that in C, arrays are passed call-by-reference but structures call-by-value. We impose the following extra static restrictions upon Core C.

Declarations Only pointers to arrays are allowed. Pointers to base type variables or structures are *not treated* in Core C.

Functions Procedures are not allowed, that is, all functions must explicit return a value via a `return` statement.[2] Note, there must exist a return statement for all possible exectution paths through functions—also "dead" branches (which often will be considered by partial evaluators).

[2]In practise, this requirement is fulfilled during transformation from C to Core C

Statements Circular data structures are not allowed (at specialization time). In assignments where the lefthand side expression is a pointer (*e.g.* `int *p`), only the forms `assign p =` *exp* and `assign p[0] =` *exp* are allowed.

Expressions Since only pointers to arrays are legal, the address operator must solely be applied to expressions of array type. It is illegal to take the address of a base-type variable or a structure. This will be motivated later. External functions are not allowed to side-effect through passed parameters, nor to refer to non-passed variables.

Justification of and motivation for these restrictions will be given in the following chapters.

2.1.3 Operational Semantics for Core C

The dynamic semantics of Core C is formally specified in a style resembling Structural Operational Semantics [Plotkin 1981] and Natural Semantics [Kahn 1987]. The semantics has three parts: function *invocation*, *execution* of statements, and *evaluation* of expresions. An implementation of the semantics is outlined in Section 2.3 in form of a self-interpreter.

Semantic Domains and Notation

Assume a Core C program p given. In order not clutter up the semantic rules with a program argument, we "globalize" it via the semantic function $\mathcal{P}$.

Definition 2.2 (Program)
Let the function $\mathcal{P} : FId \to Label \to Stmt$ *be defined by*

$$\mathcal{P}(f)(l_f) = \textit{The } l_f \textit{ labelled statement in function } f$$

for all $f \in \{f \mid f \textit{ is a function defined in } p\}$ *and* $l_f \in \{l \mid l \textit{ is a label in function } f\}$. □

The reader may think of $\mathcal{P}$ as a two-dimensional array of functions of statements.

Definition 2.3 (Value Domain)
Let the domains Value and Store be defined as follows:

Value	=	*Int + Addr + Struct*	*A value*
Int	=	$\mathbb{N}$	*An integer*
Addr	=	$\mathbb{N}$	*An address*
Struct	=	$Value^+$	*A structure*
Store	=	$\mathbb{N} \to Value$	*The store*

where Int represent integers, Addr represent store addresses and Struct represent structures. □

The store is modelled as a map from addresses to Values. The environment $\rho : \mathrm{Id} \to \mathbb{N}$ maps identifiers to addresses.

NOTATIONAL CONVENTION 2.1
Let σ, σ' be a stores. Then $\sigma[l]$ denotes the content of location l (if defined) and $\sigma[v/l_0]$ denotes the store defined by:

$$\sigma[v/l] = \begin{cases} v & \text{if } l = l_0, \\ \sigma[l] & \text{otherwise} \end{cases}$$

for addresses l.

An array $a[i_1]\ldots[i_n]$ is represented by a pointer in store $\sigma(\rho(a))$ to a store representing $a[i_1]$ *etc.* We assume the domain of store representing $a[i_1]$ is disjunct from the domain of σ, but will leave the details unspecified. The model supports the view than an array is represented by a pointer to a chunk of memory.[3]

For notational convenience, we take l' to represent the label of the statement textually following the l labelled statement (if any).

The Semantics of Function Execution

The semantics of function execution is a transition relation from (the name of the) function to execute and a store (binding global and actual parameters) to a return value and a (possibly) modified store:

$$\vdash^{\text{FUNC}}: \text{FId} \times \text{Store} \rightarrow \text{Value} \times \text{Store} \tag{2.1}$$

where the original store may be modified during the function invocation due to side-effects on non-local variables.

An execution of a function consists of three steps: allocation of store for local variables, execution of the statements and de-allocation of store for local variables. For simplicity allocation and reclaiming of store is left unspecified.

DEFINITION 2.4 (SEMATICS OF FUNCTION EXECUTION)
Let f be a function name, ρ an environment and σ a store (representing the values of the actual parameters of f in appropiate locations). Exectution of f yields value v and store σ' iff:

$$\rho \vdash^{\text{FUNC}} \langle f, \sigma\rangle \Rightarrow \langle v, \sigma'\rangle$$

where the relation $\vdash^{\text{FUNC}}$ is defined in Figure 3. □

The label l_{first} denotes the label of the first statement in function f. The semantics for execution of a list of statements is defined below.

[3]Observe, this is not the case in reality. Here a denotes a location, not a pointer. The location is converted into a pointer when a is passed as a parameter

$$[\mathit{func}\ ] \quad \frac{\begin{array}{l}\sigma' = \sigma[\cdot/l_0, \ldots, \cdot/l_m] \\ f, \rho \circ [x_1 \mapsto l_1, \ldots, x_n \mapsto l_n] \vdash^{\text{STMT}} \langle l_{first}, \sigma' \rangle \Rightarrow \langle v, \sigma'' \rangle\end{array}}{\rho, \vdash^{\text{FUNC}} \langle f, \sigma \rangle \Rightarrow \langle v, \sigma'' \rangle}$$

Figure 3: Operational semantics for function execution

Semantics for Statement Execution

The semantics for execution of a sequence of statements (a function body) is defined as a transition relation from a label and a store to a value and a store:

$$\vdash^{\text{STMT}}: \text{Label} \times \text{Store} \rightarrow \text{Value} \times \text{Store}.$$

The final value is produced due to execution of a `return` statement.

Definition 2.5 (Statement Execution)
Let f be the index of a function, σ a store, and l the label of a statement in function f. Execution of the statements in f initiated at statement l is the value v and store σ' iff:

$$f, \rho \vdash^{\text{STMT}} \langle l, \sigma \rangle \Rightarrow \langle v, \sigma' \rangle$$

where $\vdash^{\text{STMT}}$ is defined in Figure 4, and ρ is an enviroment. □

An assignment is executed by evaluation of the lefthand side and righthand side expressions, and an updating of the store. The next statement to execute is the textually following.

A goto is simply executed by a jump to the target label.

In the case of an if statement, the test expression is evaluated and compared to zero. A jump to the then or else target is performed.

A return statement is executed by evaluation of the expression. This ends the execution of a function body.

A call statement is executed the following way. The actual parameters are evaluated and the values stored in fresh locations. The environment is updated accordingly. Then the function is executed— returning a value. The value is store in the location corresponding to the variable on the lefthand side of the call. The next statement to execute is the textually following.

Semantics for Expression Evaluation

The semantics for expressions and lefthand side expressions are as expected. The meaning of expressions is a function from an expression to a value, the meaning of lefthand side expressions is a function from an expression to an address

$$\vdash^{\text{EXP}}: \quad \text{Expr} \rightarrow \text{Value} \tag{2.2}$$

$$\vdash^{\text{LVAL}}: \quad \text{Expr} \rightarrow I\!N \tag{2.3}$$

where addresses are integers. Observe that evaluation of an expression cannot effect the store.

Definition 2.6 (Evaluation of Expressions)
Let exp be an expression, ρ an enviroment and σ a store. The meaning of exp is the value v, iff

$$\rho, \sigma \vdash^{\text{EXP}} exp \Rightarrow v$$

using the rules defined in Figure 5. Let lexp be a lefthand side expression. The meaning is the address a, iff

$$\rho, \sigma \vdash^{\text{LVAL}} lexp \Rightarrow a$$

where $\vdash^{\text{LVAL}}$ is defined in Figure 6. □

Observation 2.1 *Observe via Figure 5 that arrays possess a call-by-reference passing semantics but structures are passed call-by-value.* End of Observation

Evaluation of constants and variables is straightforward.

Evaluation of an array proceeds as follows. The two expressions are evaluated, the first yielding an address, the latter an integer (the "offset"). The result is the value at the address found by adding the offset to the address.[4]

The expression in a structure index expression evaluates to a tuple of values, cf. the Value domain. The y component is extracted.

For evaluation of unary, binary and external function call we make use of an (unspecified) operator meaning funtion:

$$\mathcal{O} : \text{OId} \cup \text{EId} \rightarrow \text{Value}^n \rightarrow \text{Value}.$$

The evaluation consist of an evaluation of the arguments and an application of the semantic function.

Operational Semantics for Core C

Definition 2.7 (Operational Semantics for Core C)
Let p be a Core C program and $d_1, \ldots, d_n$ input values to the main function. Let ρ and σ be an environment and a store, respectively, representing global variables and the input values. Then holds:

$$[\![p]\!]_C(d_1, \ldots, d_n) \Rightarrow v \quad \textit{if} \quad \rho \vdash^{\text{FUNC}} \langle f, \sigma \rangle \Rightarrow \langle v, \sigma' \rangle$$

where v is the value return by the `main()` *function.* □

[4] We have here implicit made some assumptions about the concrete representation of arrays

$$[assign\,] \quad \frac{\mathcal{P}(f)(l) = l\texttt{: assign } lval \texttt{ = } exp \quad \rho, \sigma \vdash^{\text{LVAL}} lval \Rightarrow a \quad \rho, \sigma \vdash^{\text{EXP}} exp \Rightarrow v}{f, \rho \vdash^{\text{STMT}} \langle l, \sigma \rangle \Rightarrow \langle l', \sigma[v/a] \rangle}$$

$$[goto\,] \quad \frac{\mathcal{P}(f)(l) = l\texttt{: goto } m}{f, \rho \vdash^{\text{STMT}} \langle l, \sigma \rangle \Rightarrow \langle m, \sigma \rangle}$$

$$[if\,] \quad \frac{\mathcal{P}(f)(l) = l\texttt{: if (}exp\texttt{) } m \ n \quad \rho, \sigma \vdash^{\text{EXP}} exp \Rightarrow v \quad v \downarrow int \neq 0}{f, \rho \vdash^{\text{STMT}} \langle l, \sigma \rangle \Rightarrow \langle m, \sigma \rangle}$$

$$[if\,] \quad \frac{\mathcal{P}(f)(l) = l\texttt{: if (}exp\texttt{) } m \ n \quad \rho, \sigma \vdash^{\text{EXP}} exp \Rightarrow v \quad v \downarrow int = 0}{f, \rho \vdash^{\text{STMT}} \langle l, \sigma \rangle \Rightarrow \langle n, \sigma \rangle}$$

$$[return\,] \quad \frac{\mathcal{P}(f)(l) = l\texttt{: return } exp \quad \rho, \sigma \vdash^{\text{EXP}} exp \Rightarrow v}{f, \rho \vdash^{\text{STMT}} \langle l, \sigma \rangle \Rightarrow \langle v, \sigma \rangle}$$

$$[call\,] \quad \frac{\begin{array}{l}\mathcal{P}(f)(l) = l\texttt{: call } x \texttt{ = } f'\texttt{(}e_1\texttt{, } \ldots\texttt{, } e_n\texttt{)} \\ \rho, \sigma \vdash^{\text{LVAL}} x \Rightarrow a \\ \rho, \sigma \vdash^{\text{EXP}} e_1 \Rightarrow v_1 \ldots \rho, \sigma \vdash^{\text{EXP}} e_n \Rightarrow v_n \\ \rho \circ [x_1 \mapsto l_1, \ldots, x_n \mapsto l_n] \vdash^{\text{FUNC}} \\ \qquad \langle f', \sigma[v_1/l_1, \ldots, v_n/l_n] \rangle \Rightarrow \langle v, \sigma'[\cdot/l_1, \ldots, \cdot/l_n] \rangle\end{array}}{f, \rho \vdash^{\text{STMT}} \langle l, \sigma \rangle \Rightarrow \langle l', \sigma'[v/a] \rangle}$$

$$[seq\,] \quad \frac{\rho \vdash^{\text{STMT}} \langle l, \sigma \rangle \Rightarrow \langle m, \sigma' \rangle \quad \rho \vdash^{\text{STMT}} \langle m, \sigma' \rangle \Rightarrow \langle n, \sigma'' \rangle}{\rho \vdash^{\text{STMT}} \langle l, \sigma \rangle \Rightarrow \langle n, \sigma'' \rangle}$$

$$[seq\,] \quad \frac{\rho \vdash^{\text{STMT}} \langle l, \sigma \rangle \Rightarrow \langle m, \sigma' \rangle \quad \rho \vdash^{\text{STMT}} \langle m, \sigma' \rangle \Rightarrow \langle v, \sigma'' \rangle}{\rho \vdash^{\text{STMT}} \langle l, \sigma \rangle \Rightarrow \langle v, \sigma'' \rangle}$$

Figure 4: Operational semantics for statements

$$[cst\,] \quad \sigma \vdash^{\text{EXP}} \texttt{cst}\ c \Rightarrow c$$

$$[var\,] \quad \sigma \vdash^{\text{EXP}} \texttt{var}\ x \Rightarrow \sigma(\rho(x))$$

$$[index\,] \quad \frac{\sigma \vdash^{\text{EXP}} e_1 \Rightarrow v_1 \quad \sigma \vdash^{\text{EXP}} e_2 \Rightarrow v_2}{\sigma \vdash^{\text{EXP}} \texttt{index}\ e_1[e_2] \Rightarrow \sigma(v_1 \downarrow \text{Addr} + v_2 \downarrow \text{Int})}$$

$$[struct\,] \quad \frac{\sigma \vdash^{\text{EXP}} e \Rightarrow v \qquad \sigma(v) \downarrow \text{Struct} = (v_1, \ldots, v_y, \ldots, v_n)}{\sigma \vdash^{\text{EXP}} \texttt{struct}\ e.y \Rightarrow v_y}$$

$$[uop\,] \quad \frac{\sigma \vdash^{\text{EXP}} e_1 \Rightarrow v_1 \quad \mathcal{O}(op)(v_1) = v}{\sigma \vdash^{\text{EXP}} \texttt{uop}\ op\ e_1 \Rightarrow v}$$

$$[bop\,] \quad \frac{\sigma \vdash^{\text{EXP}} e_1 \Rightarrow v_1 \quad \sigma \vdash^{\text{EXP}} e_2 \Rightarrow v_2 \quad \mathcal{O}(op)(v_1, v_2) = v}{\sigma \vdash^{\text{EXP}} \texttt{bop}\ op\ e_1\ e_2 \Rightarrow v}$$

$$[addr\,] \quad \frac{\sigma \vdash^{\text{LVAL}} e \Rightarrow a}{\sigma \vdash^{\text{EXP}} \texttt{addr}\ \texttt{\&}\ e \Rightarrow a \uparrow \text{Value}}$$

$$[ecall\,] \quad \frac{\sigma \vdash^{\text{EXP}} e_1 \Rightarrow v_1 \quad \cdots \quad \sigma \vdash^{\text{EXP}} e_n \Rightarrow v_n \quad \mathcal{O}(f)(v_1, \ldots, v_n) = v}{\sigma \vdash^{\text{EXP}} \texttt{ecall}\ f(e_1, \ldots, e_n) \Rightarrow v}$$

Figure 5: Operational semantics for expressions

$$[var\,] \quad \rho, \sigma \vdash^{\text{LVAL}} \texttt{var}\ v \Rightarrow \rho(v)$$

$$[index\,] \quad \frac{\rho, \sigma \vdash^{\text{LVAL}} e_1 \Rightarrow a \quad \rho, \sigma \vdash^{\text{EXP}} e_2 \Rightarrow v}{\rho, \sigma \vdash^{\text{LVAL}} \texttt{index}\ e_1[e_2] \Rightarrow a + v \downarrow \text{Int}}$$

Figure 6: Operational semantics for lefthand side expressions

2.2 Data Structures and Self-Representation

In this section we discuss self-representation of data structures and representation of programs. Self-representation of data structures is complicated due to their unknown (user defined) size. For instance, an arbitrary array in the subject program cannot be represented by an array of fixed size in the specializer. An important aspect to consider is the preservation of the semantics in the representation. We shall see that representation of structures is problematic. Transparent self-representation of data structures is discussed (in the framework of self-interpreters) in [Andersen 1991b].

The representation of programs is important in order to reduce the overhead of double encoding as described in the introduction. A naive program representation will cause a major memory overhead and slow down self-application considerably.

2.2.1 Encoding of Data Structures

The available data types in Core C can be described by the grammar below:

$$T ::= \text{int} \mid *T \mid [T]_n \mid T \times T \tag{2.4}$$

where "int" represent integers, $*T$ is a pointer to an object of type T, $[T]_n$ is the type of an array of size n with elements of type T, and $T \times T$ describe structures. We will often omit the subscript n on array types when convenient. Observe, the index n is dyamically determined in the case of dynamically allocated arrays.

Recall that we must define a union data type `Value` which is capable of representing data values of all types, *i.e.* there must exist an injection $T \hookrightarrow$ `Value`. The type `Value` must inevitably be a structure as this is the only data type in Core C which can represent data objects of different types.

An integer can be represented via an integer field.[5] A pointer (to an array) can be presented as a pointer to a `Value` array. However, neither a structure nor an array can be represented in Value, as their sizes can be arbitrary large. Hence, we must be content with an *external* representation.

An array of size n can be represented by a pointer to a *dynamically* allocated array of size n and type `Value`. This supports the view that an array is a pointer to a chunk of memory. For purposes to be explained later, we must, however, be able to distinguish between an "array-allocation" pointer (which always points to the beginning of the array) and an arbitrary pointer (which can point to an arbitrary position into the array).

A structure can be represented by a pointer to a dynamically allocated array, where each field is represented by an entry in the array. In the following we will implicitly assume a mapping between field names and indexes. The representation is summarized in Figure 7. The field `type_int` correspond to Int, the field `type_ptr` correspond to the fields Struct and Addr where the `tag` distinguish between these. Note the difference: in the semantic domain Value, a structure is a tuple of Values. In the computer realization of Value, it is a pointer to an array. The tag field is needed in the memory management

[5] Similarly can all base types be represented by base type fields in Value

Let the structure `Value` be declared as follows.

```
struct Value
{
   int tag;                          /*  tag field */
   int type_int;                    /*  integer representation */
   struct Value *type_ptr;    /*  array and structure representation */
}
```

Let v : `Value` be a `Value` variable. The encoding of types into `Value` is defined by:

$\mathcal{E}$: Data object $\rightarrow$ `Value` object

$\mathcal{E}[\![x$: `int`$]\!]$ $= v$, where v.`tag` = `INT` and v.`type_int` $= x$

$\mathcal{E}[\![x : *T]\!]$ $= v$, where v.`tag` = `PTR`, v.`type_ptr` $= l$, and l = *location of* $\mathcal{E}[\![*x]\!]$

$\mathcal{E}[\![x : [T]_n]\!]$ $= v$, where v.`tag` = `ARRAY`, v.`type_int` $= n$, v.`type_ptr` $= l$, where l *is pointer to an array* $[\texttt{Val}]_n$ and $\forall i = 0 \ldots n-1 : l[i] = \mathcal{E}[\![\texttt{x[i]}]\!]$

$\mathcal{E}[\![x : T_1 \times \cdots \times T_n]\!] = v$, where v.`tag` = `STRUCT`, v.`type_ptr` $= l$, where l *is pointer to an array* $[T]_n$ $\forall i = 0 \ldots n-1 : l[i] = \mathcal{E}[\![\texttt{x.i}]\!]$

where `INT`, `PTR`, `ARRAY` and `STRUCT` are integer tags.

Figure 7: Encoding of data objects in structure `Value`

routines for copying, restoring and comparing specialization time store. This is treated in the next chapter.

Reconsider the representation of structures and recall that structures are passed call-by-value but arrays call-by-reference. Assume a (self-) interpreter interpreting a function call where a structure is passed as argument. Using the representation as defined above, the interpreter will pass the *address* of the array representing the structure but *not the array* itself. Hence, the interpreter implements a call-by-reference semantics for passing of structures which is inconsistent with the definition. The problem is that structures are represented by data objects encompassing another passing semantics.

In order to correct the problem, the interpreter must explicitly *copy* structures (array, that is) each time these are assigned. Assignment between structures can either be due to ordinary assignments or parameter passing. This is very similar to the code which for example the GNU C compiler generates for structures [Stallman 1991].

2.2.2 Efficient Self-Representation of Programs

A Core C program can, a bit simplified, be viewed as a collection of function definitions where each function definition contain of a number of statements. Assume structures `Fun` representing functions and `Stmt` representing statements respectively. When a program can be reprsented by a one-dimensional array of `Fun` structures, where each `Fun`

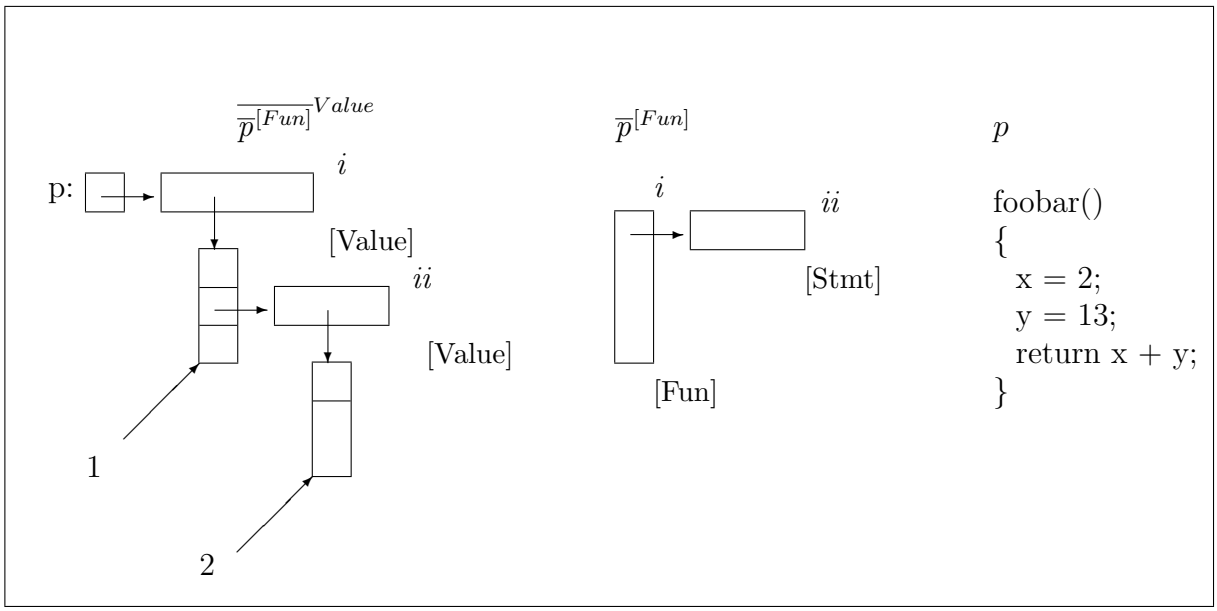

Figure 8: Data structure for program representation during self-application

structures (among other things) contains a `Stmt` array. We will not specify the concrete representation of statements and expresions—this is addressed in Chapter 7.

Let p be a program and consider the data structures living in the running version of *spec* during self-application

$$[\![spec]\!]_C(\overline{spec}^{[\texttt{Fun}]}, \overline{\overline{p}^{[\texttt{Fun}]}}^{Value}).$$

Assume that *spec* represents the subject program in a variable p. In Figure 8 the relevant data structures are illustrated, to the right the program p, in the middle the representation of p in the version of *spec* being specialized, and to the left, the representation of p as a value in the running version of *spec*.

In the version of *spec* being specialized, the program is represented as an array of `Fun` structures where each `Fun` contains a `Stmt` array. Consider the representation of this two-dimensional array in the running version of *spec*.

The `Fun` array (marked i in the Figure) is represented by a `Value` array ([Value]). As each element is a `Fun` structure, it is represented externally (marked 1 in the figure). As one of the fields in `Fun` is an array of `Stmt` two new indirections occur (marked ii and 2 in the figure). Hence, the two-dimensional array is blown up to a four-dimensional array by the double encoding, and thus, each time a statement is referenced, two extra indirections are necessary.

The same observation was made by Launchbury when he designed a data representation for a subset of LML [Launchbury 1991] and it was further investigated in [De Niel *et al.* 1991].

A better representation would be one with no indirect representation of the `Fun` and `Stmt` structures. We can achieve this goal by *including* `Fun` and `Stmt` as direct representable structures in `Value`. That is, introduce `Fun` and `Stmt` as new base types in

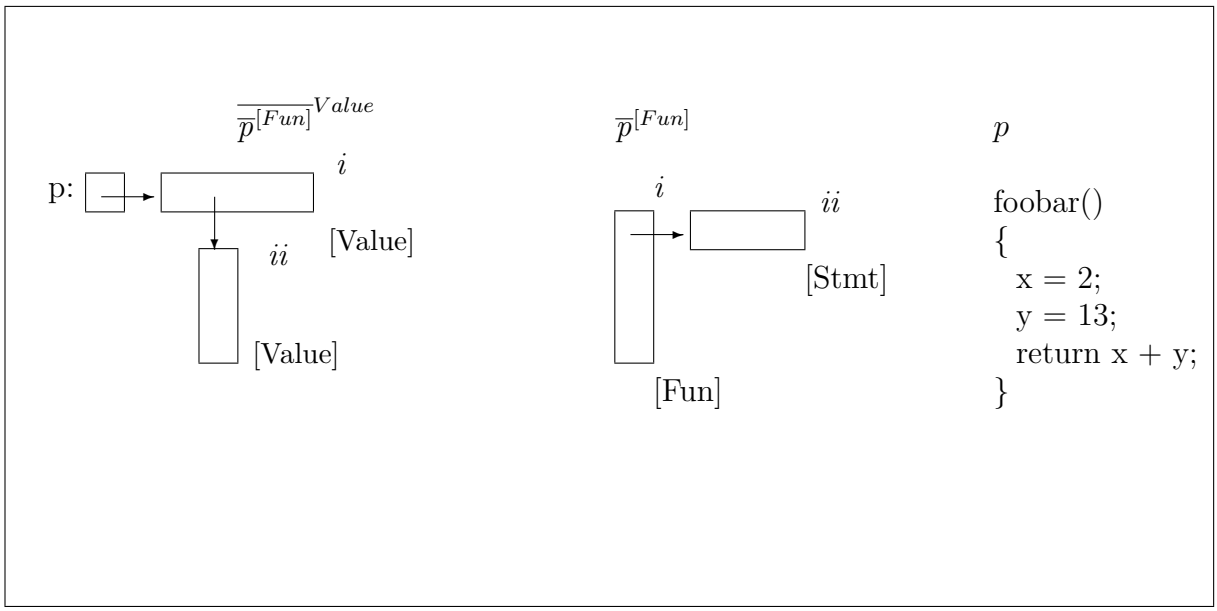

Figure 9: Data structure for program representation during self-application

Core C. Introducing two new fields `type_fun` and `type_stmt` in the `Value` structure, and extending the data type encoding function $\mathcal{E}$ accordingly, we arrive at the wanted representation.

$\mathcal{E}[\![x \ : \ \texttt{Fun}]\!] = v$, where $v.\texttt{tag} = \texttt{FUN}$ and $v.\texttt{type\_fun} = x$
$\mathcal{E}[\![x \ : \ \texttt{Stmt}]\!] = v$, where $v.\texttt{tag} = \texttt{STMT}$ and $v.\texttt{type\_stmt} = x$

The consequence of this is illustrated in Figure 9.

Observe how the two-dimensional array in the version of *spec* being specialized now is represented by a two-dimensional array in the running version of *spec*. The overhead is now solely the projection of a `Stmt` out of `Value` which is tolerably.

2.3 Self-Interpretation in Core C

In this section we sketch the bare core of a self-interpreter for Core C (in C, cf. Section 2.1). Our aim is not to present a "runable" program, but to outline the implementation of the dynamic semantics described in the previous section, and address in detail the representation of the store. The core of the self-interpreter can be found in the specializer developed in the next chapter.

For representation of programs we assume the `Fun` and `Stmt` structures as given above. We use a library of syntax function for extraction parts of *e.g.* a statemt. For example, the external function `cmix_assign_index_exp()` may return the n'th index expression in an assignment with an array expression on the lefthand side. The addressing into the store `store` can hence be programmed as follows, using the pointer `lval` to "chase" pointers.[6] Let the lefthand side expression be `a[`e_1`]`$\ldots$`[`e_n`]`.

[6]We have on purpose omitted all type tag operations

```
lval = &store[ index of  a];
for (i = 0; i < n; n++)
   lval = &lval[0][ value of cmix_assign_index_exp(i)];
lval[0] =  value of righthand side;
```

First the `lval` pointer is assigned the address of the variable `a`. Then, in the loop, the `lval` pointer is adjusted accoring to the index expressions. The idiom `lval[0] = *lval` is used for doing an indirect assignment (as the star operator not is included in Core C on a lefthand side).

In Figure 10 an outline of a self-interpreter for Core C is given—the *interpreter loop*. The mode of operation should be obvious. The self-interpreter is similar to the semantic definition of Core C. Note the function `assign`. Its purpose is to copy the content of a structure at assignments which is necessary due to the representation of these. Evaluation of expressions is done via the external function `eval_exp()`.

2.4 Summary

The kernel part of C, Core C, has been introduced and formally defined. Core C can be seen as a textual representation of the flow-graph of an C program, and includes base type variables, structures, multi dimensional array, and a restricted use of pointers. The meaning of Core C was given by the means of an operational semantics.

We addressed the problem of self representation of data structures, and gave an encoding into a union type `Value`. Sematically aspects of the representation were also considered, and in particular the representation of structures. The representation of programs was improved by the inclusion of `fun` and `Stmt` as new base types in Core C.

Finally we showed the outline of a self-interpreter for Core C.

```
Fun pgm[];                  /*  program  */
Value *gstore;              /*  global store */

/*  execute function fun_index on parameters pstore */
Value
exec_func(int fun_index, Val *pstore)
{
    Value *lstore;              /*  local store */
    Value *store;               /*  actual parameter store */
    Value *lval;                 /*  store pointer */
    int label;                    /*  program point */

    label = 1;
    while (1)
       switch ( kind of pgm[fun_index][label])
       {
       case ASSIGN: /*  assignment statement */
         lval = &{p,l,g}store[ variable index];
         for (i = 0; i < # indexes ; i++)
           lval = &(*lval)[eval_exp( i'th index exp, {p,l,g}store);
         *lval = assign(eval_exp( exp, {p,l,g}store));
         label += 1;
         break;
       case GOTO: /*  goto statement */
         label =  goto label;
         break;
       case IF: /*  if statement */
          if (eval_exp( if-test, {p,l,g}store)
             label =  then label;
          else
             label =  else label;
          break;
       case CALL: /*  call statement */
          for (i = 0; i < # parameters; i++)
             store[n] = assign(eval_exp( i'th parameter, {p,l,g}store));
          {p,l,g}store[ variable  index] = assign(exec_func( function index, store));
          label += 1;
          break;
       case RETURN: /*  return statement */
          return eval_exp( return expression, {p,l,g}store);
       }
}
```

Figure 10: Self-interpreter for Core C (with some syntactic sugar)

Chapter 3

Specialization of Core C

In this chapter we study specialization of the Core C language. The basic techniques are *program point specialization*, *reduction* of dynamic expressions, *symbolic* evaluation of static expressions and various other optimizations such as function unfolding. Program point specialization include *function specialization* and *specialization* of target points for jumps. Due to the imperative nature of Core C, *i.e.* the presence of states and destructive effects, program specialization is considerably more complicated than in a purely functional world. For example, it must be guaranteed that side-effects happens in the same order both at specialization time and in the residual program, as in the subject program. Furthermore, since the specializer operates on run time configuratations of the subject program, problems with dangling pointers, aliases and sharing emerges. The wish for self-application also imposes severe restrictions such as good binding time separation and upon the use of external functions.

The aim of this chapter is an analysis of the new problems; we believe, however, that many of the observations may be valuable in the development of partial evaluators for other languages, *e.g.* object oriented languages with states. The next chapter present a formal specification of the Core C program specializer formally defined via a *two-level* operational semantics.

As described in Section 1.3, partial evaluation can be considered as a non-standard semantics of the language. Loosely speaking: in the non-standard semantics, a static statement is executed but code is generated for a dynamic one. This means that all statements (and other constructs: variable declarations and expressions) must be classified as begin either *static* or *dynamic*. As advocated in the previous chapter we apply an off-line strategy where the binding time separation is done a priori to the actual specialization. In Chapter 5 an automatic binding time analysis for Core C is developed; for now we will assume all variable declarations (indicating whether a variable holds static or dynamic values), expressions, statements and functions (indicates whether a function is completely static) have been annotated in some unspecified manner. In the next chapter the annotation is made precise in form of a *two-level* syntax for Core C, and additional *well-annotateness* rules are imposed.

Assignment of binding times can either be *mono-variant* or *poly-variant*. In a mono-variant binding time separation, every construct, for instance a variable declaration, pos-

sess one and only one binding time. Poly-variance allows a construct to assume several binding times depending on the context. For example, with poly-variance, a function parameter may be static at some calls but dynamic at others. When a mono-variant separation is used, the parameter will be marked "dynamic" at all calls. Clearly, a poly-variant binding time assignment may give better specialization but it also complicate the specialization process. If the poly-variance is solved by the means of program transformations before the specialization, the techniques in this chapter can be applied to poly-variant specialization. We will make use of mono-variant binding time assignments, but discuss poly-variant specialization and binding time analysis in Chapter 8.

In this chapter we assume knowledge of the basic techiques of program specialization corresponding to the references [Jones *et al.* 1989], [Gomard and Jones 1991a]. For full benefit these or similar literature should be known. The most well-established terminology will be used without definition or explanation, see for example the latter reference.

The chapter is organized as follows. First, in Section 3.1 we consider function specialization in Core C. New aspects are treatment of global variables when non-local side-effects are possible, pointers and aliases due to call-by-reference passing of parameters, and sharing of generated functions. We shall see that in some cases conservative choices must be made in order to preserve the semantics. In the last part of this chapter we outline several improvements which may compensate for some of the restrictions.

From the external view of function specialization, Section 3.2 examine the specialization of the list of statements making up a function body. Central concepts are *dynamic basic blocks*, *transition compression* and *code generation*. During the specialization of a function body, several versions of the specialization time store may exists. Care must be taken with respect to pointers and aliases, *e.g.* it must be prevented that a pointer refer to the "wrong" copy of the specialization time store. We address the management of the store and provide a solution in form of a copy/restore scheme as known from copying collectors.

Section 3.3 is devoted to *partially static data structures*. A data structure is said to be *partially static* if it contains dynamic values, but parts of the structure is statically known. An example is an array containing dynamic elements which is overall statically indexed. The idea is to *split* the structure during specialization making use of the static information.

Preferable, a specializer should be capable of specializing away a complete layer of interpretation. When this is the case, the specializer is said to be *optimal*. In Section 3.4 we consider optimality of the techniques described, and find that several "tricks" are needed to achieve optimality. These are described, and in Chapter 7 we demonstrate that the specializer is able to remove a complete layer of interpretation by specializing the Core C self-interpreter to a program.

In the last section we discuss optimizations and advanced techniques. This includes specialization to live variables, function unfolding, and expression folding. In practical experiments some of these optimization has been found to be very important, for example, live variable information may typical halve the size of residual programs. In Chapter 7 we will address some of the "tricks under the carpet"; in this chapter we concentrate on basic techniques.

3.1 Function Specialization

Assume all functions have been classified as being either static or *residual*. A function is called *residual* if it contains dynamic statements. A residual function can be *specialized* or *unfolded*. Unfolding consist in replacing the call with (a specialized version of) the body; for now we will not consider unfolding but assume all functions are specialized. By *function specialization* we mean the generation of a new function, specialized with respect to the static values. The term "residual" is often overloaded to also denote the specialization time generated function.

Function specialization is complicated by the two-level scope of variables: global variables and parameters/local variables. Furthermore, call-by-reference passing of parameters give rise to pointers and aliases which must be dealt with.

3.1.1 Referential Opacity

A function is said to be *referentially transparent* if two calls to it with the same arguments always yields the same result. It is well-known that referential transparency is not valid in imperative languages due to possible side-effects on non-local variables. In C, it is even possible to define functions with an internal state which is preserved through several calls—this facility is not, however, allowed in Core C.

Example 3.1 Consider the following program fragment and assume `foo()` is residual and must hence be specialized due to the calls in `main()`.

```
int global;  /*  A static global variable */
int main()
{
    global = 5;
    x = foo(3);
    ⟨statement S using  global⟩;
    global = 5;
    x = foo(3);
}

int foo(int z)
{
    global += 1;
    return ⟨dynamic expression⟩;
}
```

Obviously, `foo()` must be specialized with respect to both `z` and `global`, and the specialization must take place *before* the statement S is processed, otherwise `global` will assume the wrong value. End of Example

Observation 3.1 *Function specialization must be with respect to both static parameters and static global variable.* End of Observation

We will later refine this observation to obtain better results, that is, sharing of generated code.

OBSERVATION 3.2 *Function specialization must be depth-first, i.e. a function has to be specialized before the statements succeeding the call are processed.* END OF OBSERVATION

The "standard" polyvariant specialization algorithm as *e.g.* given in [Bondorf 1990b] will fail miserably in our setting. In that algorithm, a function to be specialized is first scanned and dynamic calls collected for later processing, and then the body of the function is specialized. This only works in referential transparent languages.

Consider now the second call to `foo()`. Apparently, both the `global` variable and the parameter equals the previous call to `foo()` and hence, it may look like the residual version `foo_3()` (that is, `foo()` specialized with respect to `global` $= 5$ and `z` $= 3$) can be reused. Doing this blindly is wrong, however, since the `global` variable then will possess the (static) value 5 after the call; 6 is the best!

OBSERVATION 3.3 *Sharing of specialized functions may require updating of non-local variables.* END OF OBSERVATION

Note that the (static) variables which may possess the wrong values are global variables or call-by-reference passed parameters. The problem arises because a function can "return" a value through *e.g.* a global variable—this is not possible in a purely functional language. We will come back to this problem below and present a solution.

3.1.2 Static Side-Effects under dynamic Control

Consider the program in the following example where a global variable is assigned in the branches of an `if` in a residual function.

EXAMPLE 3.2 Suppose `foo()` is a residual function and the expression in the `if` statement is dynamic. Hence the jump cannot be decided at specialization time, and both branches must be specialized.

```
int global;  /*  A static global variable */
int main()
{
    x = bar(y);
    /*  value of  global?? */
    ⟨statement S using  global⟩;
}

int foo(int z)
{
    if (⟨dynamic expression⟩)
       global = 0;
    else
```

```
        global = 1;
    return ⟨dynamic expression⟩;
}
```

After the specialization of `foo()`, the concrete value of `global` will be unknown even though it is assigned statically. End of Example

An assignment to a static variable which is dynamically determined is called *static assignment under dynamic control*.

In the on-line partial evalautor for a subset of Pascal by Meyer [Meyer 1991] static assignments under dynamic control are handled by insertion of *explicators*. In the example above, the assignments to `global` would be performed, but after the call, the binding time status of `global` would be changed to "unknown" (corresponding to "dynamic" in our terminology). Hence, the knowledge that `global` is either 0 or 1 will not be used outside `foo()`.

In the off-line partial evaluator by Nirkhe and Pugh [Nirkhe and Pugh 1992] the same strategy is applied, even when the assignments are local to the function being processed. Note that we only conside static assignments under dynamic control in called functions; static assignments in the function being processed can be handled without problems using the methods to be described in the next section.

Two possible solutions to the problem is *unfolding* and *specialization to finite many values*.

Example 3.3 Consider the subject program in Example 3.2. By *unfolding* `foo()` the following (correct) residual program could be produced.

```
int main()
{
    if (⟨reduced expression⟩)
        ⟨statement S specialized with respect to  global = 0⟩;
    else
        ⟨statement S specialized with respect to  global = 1⟩;
}
```

The body of `foo` has been unfolded into `main()` and the statements following the call moved inside the branches and specialized. End of Example

The general stategy is (formulated in Core C terms): replace a `call` statement with a copy of the called function; in that code, replace every `return` statement with a `goto` to the label following the (original) call. Note this can in fact be done before the specialization on the basic of the binding time informations.[1] Observe the poly-variant specialization algorithm to be presented below allow sharing of the code after an unfolded function if the

[1] Problems with termination of the unfolding must be addressed, through. See later

values of the side-effected variables equals at several return statements. When unfolding functions problems with variable clashes, parameter passing and non-local side-effects must be considered. This is done in Section 3.5.

Function unfolding have some drawbacks. In the worst case, all functions may be unfolded into the `main()` function destroying all modularity and sharing of residual code. Even worse, as little code sharing takes place, the residual programs may increase wildly.

The idea of *specialization to finite many values* is founded on the following key observation.

OBSERVATION 3.4 *Suppose the specialization process terminates. The static variables assigned under dynamic control can only possess finite many values.* END OF OBSERVATION

The general idea is to specialize the statements after the call with respect to each of the actually value(s) of side-effected variables, but without unfolding the called function. Code must be generated in the residual function to indicate at run-time to which version of the specialized statements to execute after the call. In the example, it must be indicated whether to execute S specialized with respect to 0 or 1. Here we face the problem that C have no facilities for "continuations", it is not possible to return a label (that is, an address to jump to), and moving the S staments to new functions (called in the residual function) is not desirable as it may exhaust the program stack.[2] Our solution is a naive simulation of the `setjmp()`/`longjmp()` functions in C, using a `switch` (or nested `if-else` statements).

EXAMPLE 3.4 Continuing the example from above, a variable `endconf` is introduced at specialization time. Its use is, at run-time, to indicate which value(s) the static side-effected variable (at specialization time) have after a function call—it acts like a continuation.

```
int endconf; /*  End configuration variable */
int main()
{
    x = bar_y();
    switch (endconf)
       {
       case 1:  /*  global = 0 */
           ⟨Statement S specialized wrt.  global = 0⟩;
           break;
       case 2:  /* global = 1 */
           ⟨Statement S specialized wrt.  global = 1⟩;
           break;
       }
}

int bar_y()
```

[2] If our language supported function pointers we could apply the same trick as used in the Spineless, tagless G-machine [Peyton Jones 1991]

```
{
    if (⟨reduced expression⟩)
        {
            endconf = 1; /*  case:  global = 0 */
            return ⟨reduced expression⟩;
        }
    else
        {
            endconf = 2; /*  case:  global = 1 */
            return ⟨reduced expression⟩;
        }
}
```

The `endconf` variable is assigned a (unique) number before each `return` statement in the called function, and this number is used to determine which code to branch to after the call. End of Example

In general, each time a `return` statement is processed in the called function, an assignment to the `endconf` variable is also generated. After the (dynamic) call, a `switch` on the `endconf` variable is made. The statements following the call is specialized with respect to the values of the side-effected variables which are collected when the called function is being specialized.

Definition 3.1 (End Configuration Stores)
The set of (specialization-time) stores representing the (final) values of the static variables is called the end configuration stores. □

Still continuing the example from above, there are two end configuration stores generated: one mapping `global` to 0, the other recording `global` to 1.

After the specialization of a function (call), the specialization time store must be updated according to the end configuration stores. Observe, the only non-local variables a function can side-effect are global variables or call-by-reference passed parameters, *i.e.* arrays. In the next chapter the updating of the store is formalized. Furthermore notice that the end configuration stores records the values needed for sharing of residual functions, cf. the discussion above. For example, in the Example 3.1, there would be one end configuration store after processing `foo()`, mapping `global` to 6.

Both the above described program transformations are unable to deal with (mutually) recursive functions. The unfolding approach fails due to non-termination of unfolding, the latter since the set of end configuration stores must be known for processing a `call` statement. Assume a function f calling itself recursively.[3] Even though it has been recorded that f is currently being specialized to the same arguments as at the new call, it is not possible just to generate a new residual call. The result of the whole specialiazation of f must be known, *i.e.* all `return` statements seen.

[3]This may perfectly well happen due to the hyper-strictness of partial evaluation in contrast to ordinary evaluation

In the partial evalautor reported in [Nirkhe and Pugh 1992], a least fixed point of the possible values are found. We do have, however, a strong suspicion that this would prevent self-application. We therefore adopt the conservative solution and require non-local side-effects in recursive functions to be dynamic.

Definition 3.2 (Specialization of Recursive Functions)
Let f be a residual, recursive function. All side-effects on non-local variables must be dynamic. □

Practical experiments has shown that this is not too restrictive. A typical use of static assignments under dynamic control arises in a non-recursive functions, for example a scanner reporting "end-of-file" by setting a flag. Such cases can be handled by our methods.

Exact determination of recursion is undecidable. In Chapter 5 we describe an approximating calculation to be used in the binding time analysis.

Notational Convention 3.1 *To the syntax of Core C we add the statement* `rcall` *for calls to (possibly) recursive functions.*

3.1.3 Functions and Call-by-Reference Parameters

Call-by-reference parameters can be created by use of the address operator `&` or by passing of arrays. In many functional languages, for instance Scheme, parameters are in fact passed call-by-reference, but as the called function cannot side-effect the data structures, this is immaterial for function specialization. However, are mutable data structures allowed, for example by via `set-car!`, the considerations in this section applies.

Consider the program fragment below which declares two arrays `a[10]` and `b[10]`, and suppose two calls to `foo()` where `a` and `b` is passed as parameter, respectively.

```
int a[10], b[10];
int foo(int *x)
{
    x[2] = 5;
    return ⟨expression⟩ + b[2];
}
```

Let `a` and `b` be static variables, and assume `a` and `b` equals in all entries. When the call `foo(a)` is met, the residual function `foo_a()` is made. At the second call, `foo(b)`, one could imagine that the already generated residual function can be reused. As easy to see, however, this would be wrong.[4]

Observation 3.5 *Function specialization to call-by-reference parameters must be with respect to both the address and the indirection of the pointers.* End of Observation

[4]Unless b[2] happens to be 5!

This observation can be compared with the functional case where *e.g.* to lists are deemed equal if they equals elementwise. The observation also implies that when two specialization time stores are compared (which is done when it is to be determined whether a dynamic function call has been seen before) it is not sufficient to say "equals" solely on the basis of pointer *addresses*—the indirection must be followed.

The above requirement is rather conservative. Sharing of residual functions can only take place when the parameter is stored in the same location, *i.e.* is the same variable. There are several ways to liberate this restriction. First, if the called function f do not modify the parameters (or any of the function called by f does) then it is safe to specialize solely with respect to the indirection. Secondly, if the function only use (and modify) local variables or parameters, addresses can be ignored. We consider pointers and aliases in Section 8.2.

3.2 Dynamic Basic Block Specialization

In this section we study specialization of the body of a function, that is, statements. First we define the notion of *dynamic basic blocks*, then we describe *poly-variant specialization* of these, and we address *code generation* for each particular kind of statement. Finally, we analyze the management of the specialization time store. Due to destructive effects and pointers, the handling of stores in the specializer is rather involved compared to partial evaluation of functional languages.

3.2.1 Control Flow Specialization

The general idea is to specialize the *control-flow* and each particular *statement* to the static values. Static transitions can be done at specialization time, *e.g.* an `if` with a static test expression; dynamic control-flow must be deferred to run-time, *e.g.* an `if` with a dynamic test. For making the control-flow explicit, we adopt the notion of *basic blocks* [Aho *et al.* 1986]. Recall that a basic block is a collection of statements such that there is one distinguished entry point (the first statement) and the last statement is a control flow statement (`return`, `call`, `goto`, or `if`). We define a *dynamic basic block* as follows.

Definition 3.3 (Dynamic Basic Block)
Let $B = B_0, \ldots, B_n$ be a set of basic blocks. If it holds that

- *there exist a dynamic transition to B_0 (or B_0 is the first basic block in the function),*
- *all the transitions between $B_0, \ldots, B_n$ are static,*
- *all control-flow statement in the leaf basic blocks are dynamic,*

then B is a dynamic basic block. □

A dynamic basic block may be considered as a "tree" of basic blocks—branches are introduced by static `if`s or `goto`s. Note that two dynamic basic blocks may overlap.

Assume a program given with a binding time separation such that every variable is classified as static or dynamic. The separation, or *division*, must fulfill the *congruence*

```
/* Specialize func with respect to store */
spec_func(func, store)
{
    /* Initialize: insert first specialization point into pending */
    ⟨Record function func is being specialized to store⟩;
    done = 0; pend = 1;
    pending[done] = ⟨l_first, store⟩;
    /* Pending loop: specialize dynamic basic blocks */
    while (done < pend)
       {
           /* Restore computation state according to pending */
           ⟨ label, store⟩ = pending[done];
           done += 1;
           /* Specialize dynamic basic block */
           ⟨Figure 12 ⟩;
       }
    ⟨Store the generated code⟩;
    ⟨Store the end configuration set⟩;
    ⟨Record that func has been specialized to store⟩;
}
```

Figure 11: Specialization of dynamic basic blocks

requirement defined in [Jones 1988], and restated in the next chapter. Let σ^S be a specialization time store and σ^D a run-time store consistent with the division.

Definition 3.4 (Specialization and Residual Program Point)
Let l be the label of the entry of a dynamic basic block, and σ^S a store consistent with the division. The program point l is called a specialization point and the tuple $\langle l, \sigma^S \rangle$ a residual program point[Jones 1988]. □

The set of residual program points is usually called the *poly-set*. The labels of the basic blocks in the residual program is precisely the labels in the poly-set. We will often consider an element $\langle l, \sigma \rangle$ as a "label" in the residual program.

The idea is to specialize all reachable dynamic basic blocks to the static values, using a list *pending* to keep track of ramaining blocks to be processed. This technique was used in [Barzdin 1988] and also applied in [Gomard and Jones 1991a]. The same basic block may get specialized to several different instances of the specialization time store—this provide the poly-variance. A drive algorithm, formulated in pseudo-C, appears as Figure 11. As input it take a function (index) `func` and a specialization time store `store` (representing both parameters and local/global variables).

The array `pending` keep track of residual program points. In the area 0–`done`-1, already processed residual program points reside. The entries `done`–`pend`-1 holds residual program points to be processed. Suppose a jump to a specialization point l is met. If the tuple $\langle l, \sigma^S \rangle$, where σ^S is the current specialization time store, exist in `pending`, when

the residual program point has already (or will eventually be) specialized, and this can be shared. We will assume the function `seenB4()` which returns true if the given residual program point is in `pending`.

3.2.2 Code Generation for Dynamic Basic Blocks

Let a dynamic basic block B to be specialized given, with entry label l and initial specialization time store σ^S. Static statement can be executed as in the self-interpreter, cf. Figure 10, code must be generated for dynamic statements. Assume given two functions:

`eval_exp`() : Expr × Store → Value

`red_exp`() : Expr × Store → Exp

which evaluates a completely static expression, and reduces a dynamic expression respectively. The processing of statements is summarized in Figure 12.

When processing a dynamic assignment, the expressions are reduced and a residual assignment added to the accumulated residual code by the `gen_assign()` function. We will return to the reduction of lefthand side expressions in the next section in connection with partially static data structures.

A dynamic `goto` causes the target label and the actual store to be inserted into the pending array, unless the residual program point already is there. In that case the code can be shared. This ends the processing of a (branch of a) dynamic basic block.

A dynamic `if` is treated in a similar manner. The expression is reduced and the target residual points inserted into pending.

In the case of a dynamic `return`,[5] the expression is reduced and a residual statement generated. There is no next statement to process.

Consider a dynamic call. The static arguments are evaluated, the dynamic arguments reduced. It is then checked whether it is the first time a call to the function with the current store is seen. In the affirmative case, the function is specialized. Then the pending list is updated with the end configuration stores for specialization of the next program point. Finally a residual call is made.

The processing of a call to a recursive call is similar, except that it is known that the called function perform no side-effects under dynamic control and hence there are no need for end configuration stores.

The code generation is formalized in the next chapter via an operational semantics.

3.2.3 Management of the Specialization time Store

The management of the specialization time store is considerably more complex in a program specializer for an imperative language than partial evaluators for pure functional languages without effects. In this section we describe some of these difficulties and outline solutions.

[5] In a residual function no static return may occur

In imperative languages with destructive side-effects, the store is naturally updated—destructively! This has several consequences, the foremost that the specialization time store must be *copied*:

- at a specialization point,
- when a store is taken out of the pending list,
- before a function is specialized.

Suppose a dynamic `if` is processed. Then the store must be copied providing a copy for each of the two branches. When a residual program is taken out of the peinding list, *i.e.* a program point and a store, the store must be copied—the original store is needed in order to determine "seen before". Of similar reasons, the store must be copied when a function call is specialized in order to test for already specialized functions.

For copying the store, a tagged memory representation is needed. For example, the size of arrays must be known as well as the size of structures. Furthermore, general pointers must be distinguished from array declaration pointers. When a store is copied, array pointers shall be followed, but general pointers not. Assume for instance `a[10]` is an array and `p = &a[0]` is a pointer. If both `a` and `p` were followed, two copied of `a` would be created and the aliasing between `a` and `p` disappeared. This is the difference between the types `[T]` and `*T`.

Example 3.5 Consider the following program which illustrates a "dangling pointer" problem in connection with copying of the store.

```
int foo()
{
     int a[10]; int *p;
     ⟨statements⟩;
}
```

Suppose `p` and `a` is aliased, and that the store is to be copied. When `a` is followed and the array copied, but `p` is not. However, if the the pointer `p` is blindly copied, it will point at the wrong location. A wrong run-time configuration has appeared. End of Example

Two ways around this problem is either to adjust the pointer `p` to point to the copy of `a`, or to use a copy/restore memory scheme. In the latter case, there is a *working* store (where all the computations take place), and a set of copied of an actual working store. When the store must be copied (it can then only be the working store), it is copied to fresh locations. When it is to be reused, it is *restored* in the working store.

The advantages with this procedure are that adjustment of pointers is avoided, it is easy to compare a working store with a copied store, for instance to determine "seen before", and the copy operation in itself is fast. The drawback is that a lot of memory is copied each time. In our experiments, this has caused no problems, though.

```
while (label)
  switch (⟨kind of statement  label⟩)
    {
    case ASSIGN: /*  Static assignment */
        ⟨As in self-interpreter⟩;
    case _ASSIGN_: /*  Dynamic assignment: lval = exp */
        gen_assign(red_exp( lval, store), red_exp( exp, store));
        label += 1; break;
    case GOTO: /*  Static goto */
        ⟨As in self-interpreter⟩;
    case _GOTO_: /*  Dynamic goto:  goto lab */
        if (!seenB4( lab, store)
           pending[pend] = ⟨ lab,  store⟩, pend +=1;
        gen_goto(⟨ lab,   store⟩);
        label = 0; break;
    case IF: /*  Static if */
        ⟨As in self-interpreter⟩;
    case _IF_: /*  Dynamic if:  if ( exp) l1 l2 */
        if (!seenB4(⟨ l1,   store⟩))
            pending[pend] = ⟨ l1,   store⟩, pend += 1;
        if (!seenB4(⟨ l2,   store⟩))
            pending[pend] = ⟨ l2,   store⟩, pend += 1;
        gen_if(red_exp( exp, store), ⟨ l1,   store⟩, ⟨ l2,   store⟩);
        label = 0; break;
    case _RETURN_: /*  Dynamic return:  return exp */
        gen_return(red_exp( exp, store));
        label = 0; break;
    case CALL: case RCALL: /*  Static call */
        ⟨As in self-interpreter⟩;
    case _CALL_: /*  Dynamic call: x =  call f(e1,...,en) */
        ⟨Evaluate the static arguments⟩;
        ⟨Reduce the dynamic arguments⟩;
        if (!seen_callB4(f, ⟨static arguments⟩))
            spec_func(f, store1);
        gen_call(x, f, ⟨reduced arguments⟩);
        for (i = 0; i < #⟨end conf stores⟩; i++)
           pending[pend] = ⟨ label + 1,   store_i⟩, pend += 1;
        label = 0; break;
   case _RCALL_: /*  Dynamic recursive call: x = f(e1,...,en) */
        ⟨Evaluate static arguments⟩;
        ⟨Reduce dynamic arguments⟩;
        if (!seen_callB4(f, ⟨static arguments⟩))
            spec_func(f, store);
        gen_call(x, f, ⟨reduced arguments⟩);
        label += 1; break;
    }
```

Figure 12: Specialization of Statements

EXAMPLE 3.6 In Core C dynamic allocation of arrays is allowed via the C-Mix function `cmix_alloc_val()`.

```
Val *p;
p = cmix_alloc_val(20); /*  allocate array of 20 entries */
```

Assume the pointer `p` is static. This means the array must be copied during specialization, but as `p` possess a type $*T$, it is not followed. This is, of course, catastrophic. We have therefore introduced a new pointer declaration

```
Val dyn(p);
```

which possess the same meaning as `Val *p`, but causes the specializer to follow the indirection during copying. We ignore this issue in the rest of the presentation. END OF EXAMPLE

3.3 Partially Static Data Structures

A data structure containing dynamic values, but where parts of it is known, is called a *partially static data structure*.[6] An example of a partially static data structure is an array with dynamic elements but where all index expressions are static. Partially static data structures can be *split* into separate variables during (or before or after) specialization. Advantages gained include register allocation, removal of an indirection level, and it is important in connection with *optimality* of the specializer, cf. below.

Many kind of partially static data structures can be split. We will only consider a few cases here—these that can be split on-line, *i.e.* during the specialization. General variable splitting can be performed as an optimizing postprocess. Furthermore, we are only concerned with *correctness* and not *pragmatics*. For instance, in C, arrays are passed call-by-reference, that is, a pointer is transferred at a function call. If an array is split into separate variables, a less efficient function call may result as *e.g.* the number of registers can be exhausted and the variables must be passed via the program stack. Splitting an array with 10000 elements is clearly undesirable in most applications. In the next chapter we present a type system which captures samantically safe variable splitting.

Splitting of partial static data structures in a functional languages has been investigated by Mogensen [Mogensen 1988] and Romanenko [Romanenko 1990]. As splitting of partially static data structures often involve a raise of function's arity, it is also known under the name *arity raising*.

Originally, arity raising was proposed for obtaining optimality, but has later turned out to be a useful optimization in partial evaluators as it captures some of the rewriting steps of subject programs necessary to obtain good specialization. For instance, in a functional language, the environment in an interpreter is usually represented via a name/value association-list. When the interpreter is being specialized to a program, the values will be

[6] The usual term is "partially static structure, but for avoding confusion with the C data type "structure" we consistenly use partially static data structure

unknown making the whole environment dynamic. A traditional way around this problem has been to rewrite the program into two lists: a name list and a value list. When the partial evaluator can deal with partially static data structures, it may automatically reveal the static information from the environment. We shall see that splitting of arrays plays a similar role in the case of C.

3.3.1 Partially Static Arrays

Consider an array `int a[3]` and suppose it is overall statically indexed, that is, all index expressions are static. Then it can be split into the separate variables

```
int a_0, a_1, a_2
```

under certain conditions. It is crucial that the index expressions are static; otherwise an expression `a[`*exp*`]` cannot be replaced by the proper variable `a_1` (assuming *exp* evaluates to 1). We will solely consider splitting of statically indexed arrays.

Recall that arrays are passed call-by-reference. The following example is an example of a semantically unsafe splitting of a partially static data structure.

Example 3.7 In the program below, an array is passed as parameter to a function.

```
int main()
{
    int a[87];

    x = foo(a);
    return a[2];
}
int foo(int *x)
{
    x[2] = 13;
    return 3;
}
```

Suppose the array is split and passed as separate variables to `foo()`. Then, the assignment in `foo()` would be invisible to `main()`, and `main()` return the value `a[2]` happens to be before the call. End of Example

The problem is that by splitting the array a change from call-by-reference to call-by-value passing semantics is risked. It is *semantically* safe to split an array if it is *splittable*.

Definition 3.5 (Splittable arrays)
Let `a` *be an array. Array* `a` *is non-splittable if:*

1. *array* `a` *is passed to a function, or*
2. *array* `a` *is returned by a function, or*

3. array `a` *is passed to an external function.*

An array is splittable if it is statically indexed and not is non-splittable. (The definition will be slightly weakened later in this chapter.) □

The definition is motivated as follows. If an array is passed as parameter to a function, it is semantically unsafe to split it, cf. the above example. If an array (pointer) is returned by a function, it is obviously not (in general) possible to split it. Finally, an external function is a "black box" from the specializers point of view. The functionality of an external function is fixed and cannot be changed. Thus, it is not legal to pass separate variables when an array is expected.

On the other hand, if none of the above restrictions are violated, and the array is statically indexed, then it obviously can be split on-line. We will later relax some of the conditions in order to achieve optimality, but only in special cases where safety can be assured.

Example 3.8 Consider the following program where a pointer is assigned the first location of an array.

```
int main()
{
    int a[10], *p;
    p = a;
    *p = 87;
}
```

Suppose `a` is statically indexed and can be split. Apparently, it does not make sense to "split" the pointer `p` to the variable `p_0`; a non-existing variable. We shall see that the type system introduced in the next chapter correctly will prohibit splitting of `a`, and leave `p[0]` as a residual indirection.

Note, an assignment `p[2]` is not allowed in Core C (but the C expression can easily be transformed into legal Core C). Suppose we allow it and the array is split. Then the assignment `p[2]` would be dead wrong. Even the idiom: `p = a` correspond to `p = &a[0]` is of no help here.[7] End of Example

More sophisticated analyses may be able to spot "splittable" arrays not caught by the above definition. Useful information include the following:

- The type of variables.
- Liveness information.
- Globalization properties.

[7] We could rely on that the compiler will allocate a sequence of variables a_0, a_1, ... in consecutive memory, but this seens highly non-portable!

If the array is an array of pointers, *e.g.* a two-dimensional array, it may very well be split as the passing semantics is not violated. Static type information can detect such cases. Suppose an array is passed as parameter to a function and it is dead after the call. Then all side-effects on it will not be used any way, and it is safe to change from call-by-reference to call-by-value passing semantics. Note, however, if the array is passed further on by the called function, it may be necessary to reintroduce call-by-reference passing by means of the address operator `&`. When an array can be globalized, the need for passing it as a parameter may be eliminated, and hence restrictions one and two in the definition of splittable can be be relaxed. We will not consider these opportunities here.

We believe that it is preferable to avoid to complicate the specializer with splitting of partially static data structures except in the simple cases containted in *splittable*. This is supported by the experiences made with functional languages [Romanenko 1990].

3.3.2 Partially Static Structures

Structures[8] may be split into separate variables in some situations, that is, the field variables moved outside the structure declaration. We distinguish between two kind of partially static C structures:

- structures with both static and dynamic field variables, and
- structures consisting solely of dynamic fields

The last possibility is an instance of the first, but the difference becomes important when a structure is returned by a function. First we consider the last case.

Definition 3.6 (Splittable Structures)
Let `struct s` *be a structure type. Structure* `s` *is non-splittable if:*

1. *a variable of type* `s` *is passed to a function, or*
2. *a variable of type* `s` *is returned by a function, or*
3. *a variable of type* `s` *is passed to an external function.*

A structure is splittable if all fields are dynamic and it not is non-splittable. □

The definition is justificated as follows.

It the structure is passed as parameter to a function, and is split, a change from call-by-value to call-by-reference semantics is risked. This can happen if the structure contains an array, as arrays residing inside structures are passed call-by-value, but call-by-reference if the surrounding structure declaration is removed. If a structure is returned by a function, it is obviously not possible to split the structure if the function still has to return all the field variables. The third requirement is included for the same reason as in the case of arrays.

The first condition can be aleviated via type informations. The fields of array type can be encapsulated inside a structure (containing the array as its sole element) which

[8] by "structure" we mean in this section the C datatype struct

assures the array is passed call-by-value. Structures being returned by functions may be split by letting the functions return the fields through call-by-reference parameters. We see no good reasons to persue this, however.

We now consider the first kind of partially structures. The problem in splitting these arises when they are returned by functions. A residual `return` returning (a structure with) the dynamic fields should be generated, but at the same time the static fields are needed for further processing. One possible solution is to apply program transformations to the subject program before the specialization, effectively separating static and dynamic variables. This would introduce a new function returning the static part of the structure. This technique has been investigated for functional languages in [Mogensen 1989]. Another way would be to allow the specializer to return both static values and generate residual code. This may, however, give problems with self-application.

We will refrain from splitting of partial static structure when they are returned by functions.

3.4 Removal of a Complete Layer of Interpretation

It is easy to construct a program specializer which complies with the requirements in the Mix Equation. For example, the specializer can simply "memorize" the static argument(s) and do no real computation. As a criterion for the optimality of *mix*, Jones [Jones 1988] proposed the following definition.

Definition 3.7 (Optimality of Mix)
Let SelfInt be a self-interpreter and p a program. Then mix is optimal iff

$$[\![mix]\!]_C(\overline{SelfInt}^{[Fun]},\ \overline{\overline{p}^{[Fun]}}^{Val}) \simeq \overline{p}^{[Fun]} \tag{3.1}$$

The definition captures that mix can remove a complete layer of (self-)interpretation. □

Observe that the definition do not assume an arbitrary self-interpreter. In practise, the self-interpreter must be "suited" for partial evaluation for obtaining optimality. The defintion states that the residual program p' should be as efficient as the original program. It is, however, too much to expect the residual program to be more efficient than the given subject program. The simple techniques used in *mix* cannot improve *e.g.* an algorithm used in a program. Automatic program specialization is no miracle-worker!

By analyzing the outlined specializer and the self-interpreter from Chapter 2, it can be seen that the following problems impede optimality:

- The store, represented by an array (of type `Value`) in the self-interpreter, must be split into separate variables;
- The use of external function for evaluation and reduction of expressions hinders reproduction of expressions stemming from the input program to the self-interpreter;
- The types in the residual program p' are too general: they are all of type `Value`.

We address the two first items below. The latter is due to the strong typing of Core C, and this is the subject of Chapter 6. The chapter presents an *untagging analysis* with the aim of simplying the types in the residual program.

3.4.1 The Store and Array Addressing

The array representing the store in the self-interpreter must be split into separate variables, resembling the variables in the input program. Otherwise all variables in the residual program will be represented indirectly in a single variable. This implies that the array must be statically indexed, cf. Section 3.3. Hereby a representation of the store as an array of activation records is ruled out, since it unavoidably will be dynamically indexed (due to an unknown number of recursion levels). Instead an array must be allocated for each function invocation, as illustrated in Figure 10. Then the store will be statically indexed (in the first level) since the index solely depends upon the program—the variable's offset. Note that recursion then is implemented by recursion.

Consider once again the array index calculation loop from the self-interpreter in Section 2.3. The first indexing is clearly static as it is the variable's (static) offset.

Now observe that the store is passed to the external function `exec_func` for passing of the values of actual parameters. This is in conflict with the requirement in the definition of *splittable*: an array passed as parameter cannot be split since the change from call-by-reference to call-by-reference may violate the meaning of the program. However, in this particular case where is no problem: the `store` is *dead* after the call to `exec_func()`. Hence, no modification of `store` during execution of `exec_func` can be observed after the call. It is therefore safe to split the store and pass it as separate variables to (the specialized versions of) `exec_func()`.

3.4.2 Expression Reduction

For evaluation of expressions, the self-interpreter uses an external function `eval_exp()`, and the specializer uses the external function `red_exp()` for reduction of expressions. Both of these must be given the store in order to do their job. This has two unpleasant consequences.

Suppose we specialize the self-interpreter to a program p. When, in the self-interpreter, the following code fragment appears, dealing with reduction of expression e in p:

$$\texttt{eval\_exp}(e, \text{store}_{SelfInt})$$

where $\text{store}_{SelfInt}$ is the store in the self-interpreter. First, as the store is (partly) dynamic, the call to `eval_exp()` must be suspended. Secondly, since the store appear as argument to an external function, it cannot be split.

In the specializer, the code interpreting the above piece of code is:

$$\texttt{red\_exp}(\texttt{eval\_exp}(e, \text{store}_{SelfInt}), \text{store}_{spec})$$

where store_{spec} is the store in the specializer. The expression e is completely static, but the values of the variables in e are unknown. Since `eval_exp()` essentially is a "self-evaluator", the desired result is e with all variables replaced with the ones used in the self-interpreter.

This can be achieved by making the external function call `eval_exp()` *known* to the specializer (`red_exp()`), and let it perform the wanted renamings inside the expression.

EXAMPLE 3.9 Let $e = \texttt{x} + \texttt{y}$ and assume `x` is represented in location 0 and `y` in location 1 in store $\text{store}_{SelfInt}$. Reduction of the call to `eval_exp()` yields

```
store[0] + store[1]
```

before splitting of the `store`, and

```
store_0 + store_1
```

after. END OF EXAMPLE

In Chapter 7 we demonstrate removal of a complete layer of interpretation by specializing the self-interpreter from Chapter 2 to a program.

3.5 Optimizations and Advanced Techniques

In this section we briefly discuss further improvements and optimizations. When self-application is wanted, it is preferable to keep the specializer simple. This is in conflict with advanced optimizations. A fruitful solution is to perform the optimizations in a separate post-phase on the residual program. We consider specialization to *live variable* (analysis done prior to specialization, information used during it), folding/unfolding of statements and functions (post-phase), and the related problem of sharing/code-duplication.

3.5.1 Static Live Variables

In imperative programs of a certain size, it is often the case that most of the variables are (locally) dead. This is problematic in connection with the outlined specializer, as it take *all* the static variables into account when deciding whether to specialize of share. Consider for example the following typical loop:

```
    int guard;
L:  if (⟨condition⟩) goto M else goto N;
M:      guard = ⟨guard expression⟩;
        switch guard
            {
            case 1: ...; break;
            ...
            case n: ...; break;
            }
        goto L;
N:
```

Assume `guard` (static) is initialize to 0 by default, and that *condition* is dynamic. Then the conditional (`L`) will be specialized with respect to `guard` $= 0$. At the `goto L` statement, the `guard` will possess another static value, and hence a new residual program point $\{\langle L, [\texttt{guard} \mapsto n]\rangle\}$ generated. Apparently, `guard` is dead at `L` and hence the residual program point should be shared.

The information needed is, at every specialization point, the set of live variables. This can easily be computed using well-known flow-analysis techniques [Aho *et al.* 1986]. The usefuylness of liveness information was discovered in [Gomard and Jones 1991a] in the setting of a flow-chart language.

Experiments with the implementation of the C specializer shows that liveness information is important. Typical, the size of the residual programs are halved, for instance due to too much specialization of a loop, as illustrated above. Liveness detection is also essential for removal of a complete layer of interpretation.

3.5.2 Folding and Unfolding

So far we have assumed all functions to be specialized. Alternatively, a (residual) function may be unfolded. The available data structures in Core C makes on-line unfolding of functions cumberstone, and we have hence decided to postpone all unfolding to a separate postphase.[9]

One could fear that long residual program consisting of "trivial" functions could be generated but experiments shows that this is seldom the case. Furthermore, the C compiler is good at performing such optimization so the lack of unfolding do not have any major impact on the efficiency of the residual programs.

Folding of statements is, on the contrary, required if "readable" programs are desired. Consider the result of specializing an array indexing `a[`e_1`]...[`e_n`] = `e where the expressions e_i are dynamic.

```
lval = &store_4;
lval = &lval[0][e'_1];
...
lval = &lval[0][e'_n];
lval[0] = e';
```

where 4 is the location of `a`. These statements may safely be folded into the single statement

```
store_4[e'_1]...[e'_n] = e';
```

which is what we want. Such foldings can be done in a postphase optimization.

In the general case, folding of statements must be done under preservation of execution order of function calls. It is *unsafe* to swap two function calls. Standard techniques for flow-analysis can be used to detect safe folding. Note that if side-effects are allowed in expressions, *e.g.* `getc()`, folding of statements becomes substantially more complicated. This issue is also addressed in the partial evaluator Similix for a subset of Scheme which includes some "global" side-effects [Bondorf and Danvy 1990].

[9] In a new implementation this problem has been overcome

3.5.3 Code Duplication and Sharing

As briefly mentioned earlier, poly-variant specialization of dynamic basic blocks may result in a lot of code duplication, especially if the basic blocks are large. A way to prevent this is to insert extra specialization points manually[10].

Code duplication could be spotted in a separate post optimization analysis. However, we believe it is not worth the effort, since a lot of time has already been put into data flow optimizations in C compilers. We see no reason to do all the work all over again!

Other analyses such as common subexpression elimination, dead code elimination *etc.*, are also useful to apply to residual programs. However, as these analyses already are incorporated in most C compilers, it does not seem worth the trouble to do it separately.

3.6 Summary

In this chapter we studied specialization of imperative languages, and in particular problems occuring due to the presence of side-effects and both local and global variables.

We gave solutions to the problem of static side-effects under dynamic control, considered specialiation of functions and statements, and the management of the specialization time stores. Conditions for specialization time splitting of partially static data structures was given under consideration of the semantics of the language.

Finally we addressed the problem of optimality, and showed how the C specializer can be improved to specialize away a complete layer of interpretation.

[10]This technique has been used in the specializer for avoiding a useless one-time unfolding of the pending loop

Chapter 4

Two Level Core C Semantics

In this chapter we develop a formal specification of the C program specializer based on the operational semantics of the Core C intermediate language from Chapter 2. First we define a *two-level* Core C syntax where binding times are explicit in the syntax. Every construct which can be either static or dynamic appears in two versions: a specialization time version which is executed during the specialization, and a run time version which causes the specializer to generate code. Thus, specialization of (Core) C can be seen as an execution of a two-level program in a two-level semantics.

The two-level syntax does not capture all the requirements that must be fulfilled for a program to be specialized. For example, the indispensable dependency principle: "static computation may not depend on dynamic values" is not caught by the syntax. We therefore impose additional *context* rules upon the two-level syntax. A two-level program satisfying the additional semantical requirements is said to be *well-annotated*. A well-annotated program can be specialized without the specializer committing errors such as "dynamic expression in static context".

The chapter is organized as follows. First we define the two-level syntax via an BNF-grammar, and introduce a binding time shift operator *lift*.

In Section 4.2 we define a *two-level* type system over two-level expressions; we lift it to statements, functions and finally Core C programs. By the means of type inference systems, we define *well-annotatedness* of a Core C program.

Section 4.3 contains a complete operational semantics for two-level Core C. Whereas the semantics of Core C yields a value, the two-level semantics yields, when applied to a well-annotated two-level program and the static values, a residual program.

4.1 Separation of Binding Times

In the two-level Core C language, every construct which can be both specialization time and run time appears in two versions. The normal (static) version, and an underlined (dynamic) version. Examples: the static variable declaration `int x` declares `x` to be a static variable which value will be known during specialization. On the other hand, <u>`int`</u> `y` declares `y` to be a dynamic variable. Similarly, `assign` denotes a static assignment, but <u>`assign`</u> a dynamic assignment. During specialization, the first will be executed, but

code generated for the latter one. The binding times of expressions are made explicit in a similar way.

The use of the syntax to pass on binding time information to the specializer implies that each particular construct only can be assigned one binding time. Hence, the binding time assignment is mono-variant. In Section 8.2, we discuss how poly-variance can be achieved using the same basic principles.

Separation of binding times via two-level syntax was founded by the Nielsons in their Two-level Meta Language [Nielson and Nielson 1988a], and later used as the fundamental principle in Lambda-Mix [Gomard and Jones 1991b]. Recently it has been adapted to Similix [Bondorf 1992].

Definition 4.1 (Two-level Core C Syntax)
The abstract syntax of Two-level Core C is defined in Figure 13. Start non-terminal is 2CC. □

The variable declaration separate the binding times of variables. In an array declaration int a$[e_1]\cdots[e_i]\underline{[e_{i+1}]}\cdots\underline{[e_n]}$, the first i dimensions can be removed, *i.e.* the array split, but the last $i+1$ to n kept. In an structure declarations struct { ... }, the structure cannot be split, but in struct { ... } the structure is split (if any dynamic fields). The return type of functions indicate whether it is specialization time or residual.

The separatation of statements and expressions are obviuos. Observe the rcall for calls to possibly recursive functions as discussed in the previous chapter. Insertion of rcall can be done automaticlly, cf. Chapter 5.

Example 4.1 Suppose we want to specialize the well-known *power* program with respect to a static n but unknown x. Then all the computation depending upon n, that is, the loop and the decrement, can be performed at specialization time. The assignment to pow is dynamic since the value of x is unknown, and so is the final return statement. A two-level Core C (annotated) version of *power* is shown below.

```
int power(int n int x)
{
    int pow

    1: assign var pow = cst 1
    2: if (n) 3 6
    3: assign var pow = bop var pow * var x
    4: assign var n = bop var n - cst 1
    5: goto 2;
    6: return var pow
}
```

The assignment of the (static) constant 1 to pow is dynamic due to the mono-variance of variable binding time assignments.

Of obvious reasons, we will often be a bit sloppy and omit abstract syntax tags on expressions in later examples. End of Example

id	$\in$	Variable identifier	
fid	$\in$	Function identifier	
eid	$\in$	External function identifier	
label	$\in$	Label	
const	$\in$	Int	
uop	$\in$	C unary operator	
bop	$\in$	C binary operator	
2CC	::=	2vardecl* 2fundef+	*Two-level Core C program*
2vardecl	::=	vardecl | 2type *id* 2typespec	*Two-level declarations*
2type	::=	`int` | 2type '*' | `struct` *id* '{' 2vardecl+ '}'	
2typespec	::=	typespec | 2typespec '[' *const* ']'	
2fundef	::=	fundef	*Two-level functions*
	|	2type *id* '(' 2vardecl* ')' '{' 2vardecl* 2stmt* '}'	
2stmt	::=	stmt	*Two-level statements*
	|	*label* ':' `assign` 2lval '=' 2exp	
	|	*label* ':' `return` 2exp	
	|	*label* ':' `goto` *label*	
	|	*label* ':' `if` '(' 2exp ')' *label label*	
	|	*label* ':' `call` *id* '=' *fid* '(' 2exp* ')'	
	|	*label* ':' `rcall` *id* '=' *fid* '(' 2exp* ')'	
2exp	::=	exp	*Two-level expressions*
	|	`lift` exp	
	|	`var` *id*	
	|	`index` 2exp '[' 2exp ']'	
	|	`struct` 2exp '.' *id*	
	|	`addr` & 2exp	
	|	`uop` uop 2exp	
	|	`bop` 2exp bop 2exp	
	|	`ecall` *eid* '(' 2exp* ')'	
2lval	::=	lval	*Two-level left-expressions*
	|	`var` *id*	
	|	`index` 2lval '[' 2exp ']'	

Figure 13: Abstact syntax of two-level Core C (see also Figure 2)

Consider again the assignment `pow = 1` in the example above. The assignment is dynamic due to `pow`, but the constant is clearly static. This phenomenon is referred to as static computation in dynamic context.

In order to make such shifts in binding times explicit, we introduce a *lift* operator (in concrete syntax we use the at-sign @ to denote lift). Intuitively, in the two-level language, the lift operator represent a shift from *evaluation* to *reduction*, or operational speaking, the meaning of `@` *exp* is "evaluate the expression and built a residual constant of the value".

It is not possible to apply the lift operator to an arbitrary expression. For example, applying lift to a pointer would leave a (specialization time) address in the residual program; applying lift to an array should (preferable) yield the array as a constant, but this is not possible in C. A similar situation can be found in Similix [Bondorf 1990a]. In the higher order partial evaluator, it is not possible to apply lift to a closure as a residual function cannot be built from a closure.

Definition 4.2 (Use of Lift)
The lift operator `lift` *may only be applied to base type values.* □

Example 4.2 Reconsider the *power* example from the previous subsection. With the lift operator at hand, the proper two-level assignment of 1 to `pow` is

```
assign var pow = lift 1
```

since the static constant 1 appears in a dynamic context. End of Example

4.2 Well-Annotatedness

Some syntactically correct programs cannot be specialized. For example, it is not captured by the two-level syntax that a lift operator is necessary in the *power* program. In this section we impose some context requirements over two-level program ending up with a definition of *well-annotatedness*. Well-annotated programs can be specialized without errors such as missing lift operators.

The binding time of an expression can be seem as the *type* of the expression in the two-level language. Hence, we introduce two-level type systems over two-level expressions, statements and functions. Well-annotatedness is then a matter of *well-typing* in the two-level type system, or put another way, a two-level program is well-annotated if it type-checks in the two-level type system.

Well-annotatedness is a matter of *correctness*: if a program is *well-annotated* the specializer is *guaranteed* to "behave well" when applied to it. With suitable definitions, we will finally in this section adapt Milner's theorem [Milner 1978]:

Well-annotated Programs Do Not Go Wrong.

Specialization of a program may fail for other reasons, though. For instance, the specializer may enter a completely static loop. We consider the *termination problem* in Chapter 5.

This correspond closely to the safety of static typing. One is guaranteed that a boolean does not show up where an integer is expected, but division by zero can still happen.

The conditions formulated in this section are purely *checking* rules. They are not constructive. The problem of deriving a *well-annotated* two-level Core C program, given a Core C program and binding times for the goal parameters, is the problem of doing *binding time analysis*. This is the subject of Chapter 5.

4.2.1 Variable Division

For notational convenience we assume all variable names different in the rest of this section. When referring to the parameters in function f, we will often use the notation f-x_1, ..., f-x_n (assuming n parameters).

Definition 4.3 (Binding Time Types)
The abstract syntax of a two-level type $\mathcal{T}$ is given by:

$$\mathcal{T} ::= S \mid D \mid [\,\mathcal{T}\,] \mid \mathcal{T} \otimes \mathcal{T} \tag{4.1}$$

The base types S and D are also called ground types. A type assignment τ is a map from program variables to two-level types:

$$\tau : Id \to \mathcal{T}$$

A function type assignment π is a map from function names to two-level types:

$$\pi : FId \to \mathcal{T}.$$

A type T in which D does not occur is called a static type (not to be confused with the ground type S). □

We will freely use the product constructor $\times$ to describe the arguments of functions, operators and external functions.

The interpretation of the two-level types is as follows. The ground types S and D denotes a completely static base type constant, a completely dynamic value, respectively. The two-level type constructor $[\,\mathcal{T}\,]$ denotes a *statically* indexed arrays with elements of type $\mathcal{T}$. A (possibly) dynamically indexed array is described by the type D. Finally, the two-level type constructor $\otimes$ describes splittable structures. Structures which cannot be split at specialization time is given a D type ("completely unknown thing").

A *type assignment* defined on all program variables in the program, correspond to a *division* in [Jones 1988]. Given a division, an assignment of binding time types to all expressions can easily be made. A function type assignment describes the binding time of the values functions return (the parameters are described by the variable assignment). Observe, it is contained in the definition that mono-variant binding times are assigned to functions, *i.e.* a function cannot both return a static and dynamic value.

For convenience, we assume the *operator assignment*:

$$\gamma : \mathrm{OId} \cup \mathrm{EId} \to \mathcal{T}$$

describing the *static* binding time of operators and external functions. For instance, $\gamma(+) = S \times S \to S$ as "+" takes two static (integers) and deliver a static (integer).

Given a variable -, a function - and an operator type assignment, a *binding time typing* is a judgement of form

$$\tau, \pi, \gamma \vdash e : T$$

which says that the expression e possess the two-level type T in the context of τ, π and γ.

A variable type assignment τ_0 defined (only) on the parameters of the main functions, is called an *initial binding time assignment*, or an *initial division*. An assignment τ which agree with τ_0 on the parameters of the main function is said to *respect* the initial division.

In the rest of this section we assume an initial division τ_0 given.

4.2.2 Well-Annotatedness of Expressions

Given a variable and an operator type assignment, well-annotatedness of expressions is expressed through a type inference system. The complicating factor is the notion of *splittable*. Recall, the type $[T]$ describes an array statically indexed with elements of type T. We want to assign the array type iff the array is splittable. For example, assume `x` is non-static, an array expression `x[`e`]` should be assigned type $[T]$ iff `x` can be split; D otherwise.[1]

Definition 4.4 (Well-Annotatedness of Expressions)
Let τ be a type assignment which respects the initial division τ_0, and γ an operator type environment. The two-level expression e is well-annotated iff there exists a two-level type T such that

$$\tau, \gamma \vdash^{\mathrm{EXP}} e : T$$

using the inference system given in Figure 14. □

The rules can be explained as follows.

A constant is always static, and a variable possesses the type that the type assignment maps it to. The lift operator applied to a static expression yields a dynamic expression.

If the left-expression e_1 in an index expression $e_1[e_2]$ possess a two-level type $[T]$, and the index e_2 is static, then the indexing can be performed (*i.e.* the expression evaluated or reduced) giving an expression of type T. Otherwise all the subexpressions must be dynamic.

Similarly holds for structure indexing. If the expression e in $e.y$ has a structured type, the indexing can be performed. Otherwise it must be residual (the "internal" of the structure is unknown, *i.e.* it cannot be split).

Consider the address operator `&`. There is no static rule for it, implying all applications are effictively suspended to run time. The reason for this is technical. If applications of

[1] Due to well-typeness, the only alternative to $[T]$ is D

$$\tau, \gamma \vdash^{\mathrm{EXP}} \texttt{cst}\ c : S$$

$$\frac{\tau, \gamma \vdash^{\mathrm{EXP}} e : S}{\tau, \gamma \vdash^{\mathrm{EXP}} \underline{\texttt{lift}}\ e : D}$$

$$\frac{\tau(x) = T}{\tau, \gamma \vdash^{\mathrm{EXP}} \texttt{var}\ x : T}$$

$$\frac{\tau(x) = D}{\tau, \gamma \vdash^{\mathrm{EXP}} \underline{\texttt{var}}\ x : D}$$

$$\frac{\tau, \gamma \vdash^{\mathrm{EXP}} e_1 : [T] \quad \tau, \gamma \vdash^{\mathrm{EXP}} e_2 : S}{\tau, \gamma \vdash^{\mathrm{EXP}} \texttt{index}\ e_1[e_2] : T}$$

$$\frac{\tau, \gamma \vdash^{\mathrm{EXP}} e_1 : D \quad e_2 : D}{\tau, \gamma \vdash^{\mathrm{EXP}} \underline{\texttt{index}}\ e_1[e_2] : D}$$

$$\frac{\tau, \gamma \vdash^{\mathrm{EXP}} e : T_x \otimes \cdots \otimes T_y \otimes \cdots \otimes T_z}{\tau, \gamma \vdash^{\mathrm{EXP}} \texttt{struct}\ e.y : T_y}$$

$$\frac{\tau, \gamma \vdash^{\mathrm{EXP}} e : D}{\tau, \gamma \vdash^{\mathrm{EXP}} \underline{\texttt{struct}}\ e.y : D}$$

$$\frac{\tau, \gamma \vdash^{\mathrm{EXP}} e : D}{\tau, \gamma \vdash^{\mathrm{EXP}} \underline{\texttt{addr}}\ \&\ e : D}$$

$$\frac{\tau, \gamma \vdash^{\mathrm{EXP}} e : T_1 \quad \gamma(op) = T_1 \to T}{\tau, \gamma \vdash^{\mathrm{EXP}} \texttt{uop}\ op\ e : T}$$

$$\frac{\tau, \gamma \vdash^{\mathrm{EXP}} e : D}{\tau, \gamma \vdash^{\mathrm{EXP}} \underline{\texttt{uop}}\ op\ e : D}$$

$$\frac{\tau, \gamma \vdash^{\mathrm{EXP}} e_1 : T_1 \quad \tau, \gamma \vdash^{\mathrm{EXP}} e_2 : T_2 \quad \gamma(op) = T_1 \times T_2 \to T}{\tau, \gamma \vdash^{\mathrm{EXP}} \texttt{bop}\ op\ e_1\ e_2 : T}$$

$$\frac{\tau, \gamma \vdash^{\mathrm{EXP}} e_1 : D \quad e_2 : D}{\tau, \gamma \vdash^{\mathrm{EXP}} \underline{\texttt{bop}}\ op\ e_1\ e_2 : D}$$

$$\frac{\tau, \gamma \vdash^{\mathrm{EXP}} e_1 : T_1 \quad \ldots \quad \tau, \gamma \vdash^{\mathrm{EXP}} e_n : T_n \quad \gamma(f) = T_1 \times \cdots \times T_n \to T}{\tau, \gamma \vdash^{\mathrm{EXP}} \texttt{ecall}\ f(e_1, \ldots, e_n) : T}$$

$$\frac{\tau, \gamma \vdash^{\mathrm{EXP}} e_1 : D \quad \ldots \quad \tau, \gamma \vdash^{\mathrm{EXP}} e_n : D}{\tau, \gamma \vdash^{\mathrm{EXP}} \underline{\texttt{ecall}}\ f(e_1, \ldots, e_n) : D}$$

Figure 14: Well-annotatedness rules for expressions

the address operator are allowed at specialization time, static pointers to the middle of arrays can be created. This complicates the comparison of specialization time stores, but a more sophisticated memory representation may alleviate this restriction.[2] We discuss this in Chapter 8.

The static rules for unary, binary and external application is as in ordinary typing. Note the dynamic rules require all arguments to be dynamic (D) is just one of the arguments are. This is in agreement with the requirement in the definition of splittable.

EXAMPLE 4.3 Recall that in order to remove a complete layer of interpretation, it is necessary to split the store-array and to reduce "inside" calls to the external function `eval_exp()`, cf. Section 3.4. This can be formalized via the special inference rule for `eval_exp()`:

[2]The problem is that the size of the array a pointer points to must be known in order to do a comparison

$$\frac{\tau(v) = T}{\tau, \gamma \vdash^{\text{LEXP}} \texttt{var}\ v : T} \qquad \frac{\tau(v) = D}{\tau, \gamma \vdash^{\text{LEXP}} \underline{\texttt{var}}\ v : D}$$

$$\frac{\tau, \gamma \vdash^{\text{LEXP}} e_1 : [T] \quad \tau, \gamma \vdash^{\text{EXP}} e_2 : S}{\tau, \gamma \vdash^{\text{LEXP}} \texttt{index}\ e_1[e_2] : T} \qquad \frac{\tau, \gamma \vdash^{\text{LEXP}} e_1 : D \quad \tau, \gamma \vdash^{\text{EXP}} e_2 : D}{\tau, \gamma \vdash^{\text{LEXP}} \underline{\texttt{index}}\ e_1[e_2] : D}$$

Figure 15: Well-annotatedness for left-expressions

$$\frac{\tau, \gamma \vdash^{\text{EXP}} e : S \quad \tau, \gamma \vdash^{\text{EXP}} \mathit{store} : [D]}{\tau, \gamma \vdash^{\text{EXP}} \underline{\texttt{ecall}}\ \texttt{eval\_exp(}e\texttt{,}\ \mathit{store}\texttt{)} : D.}$$

The rule contain two things: first, the store is not made dynamic defiance it is passed as parameter to an external function. Secondly, the store is an array of type $[D]$, and can hence be split. An additional rule for the `exec_func()` function in the self-interpreter must be added in order to prevent the store from being dynamic due to these calls. We will not dwell on this. End of Example

Definition 4.5 (Well-Annotatedness for Left-expressions)
Let τ be a type assignment which respect the initial division τ_0, and γ an operator type assignment. The two-level left-expression e is well-annotated iff there exists a two-level type T such that

$$\tau, \gamma \vdash^{\text{LEXP}} e : T$$

using the type inference system given in Figure 15. □

The rules are as the rules for righthand side expressions.

4.2.3 Well-Annotatedness of Statements

A statement may be deemed dynamic for several reasons:

1. it depends on a dynamic expression, or
2. performing it at specialization time may be undesirable due to code explosion (*e.g.* unfolding of a loop), or
3. execution of it at specialization time may cause termination problems.

We will solely consider the first condition here. Estimation of the size of residual programs and automatic generalization (that is, reclassification to dynamic) is still an open problem [Andersen and Gomard 1992]. The problem of termination is addressed in Chapter 5 and will not concern us here. Observe, given a well-annotated program, it is always possible to obtain a well-annotated program even though more constructs are classified to dynamic.

Definition 4.6 (Execution Types)
An abstract execution type $\mathcal{U}$ *is of form:*

$$\mathcal{U} ::= C \mid R \tag{4.2}$$

The type C *means specialization time or static and the type* R *denotes runtime or dynamic.* □

Definition 4.7 (Well-Annotatedness of Statements)
Let a type assignment τ *respecting the initial division, a function type assignment* π*, and an operator type assignment* γ *be given. Let* s *be a Core C two-level statement in function* f*. Statement* s *is well-annotated if there is a* $T \in \{\mathcal{C}, \mathcal{R}\}$ *such that*

$$\tau, \pi, \gamma \vdash^{\text{STMT}} s : T$$

using the inference rules given in Figure 16. □

In a static assignment, the two sides must have the same binding time type, otherwise both must be dynamic, cf. the example below. Well-annotatedness of `goto` is trivial. In static `if`, the expression must be static.[3] If the expression is dynamic, the `if` is dynamic too. Similar holds for a `return` statement.

Consider the static `call` statement. The type of the expressions must agree with the types recorded by the variable assignment map, and they must all be static. Furthermore, the called function must return a static value, and the variable assigned must possess the same static binding time. In the case of a dynamic (non-recursive) call, we assume for notational convenience that the first k parameters are static. The rest must be dynamic (D). Finally, the function must return a dynamic value, and the variable assigned must be dynamic.

The cases for recursive calls (`rcall`) are equivalent.

Example 4.4 In the rule for well-annotatedness of a run time assignment, it is required that both the expressions are dynamic (D). Assume the rule is weakened to allow *e.g.* a type $[D]$. (Recall that this denotes the type of an array which will be split at specialization time.) During the specialization, an assignment

```
assign a = b
```

of an array `int b[10000]` would have to be replaced with the sequence of assignments

```
assign a_0 = b_0
assign a_1 = b_1
...
assign a_9999 = b_9999
```

[3] We here require a S type ruling out for instance if(a) ... where a is an array

$$\frac{\tau,\gamma \vdash^{\text{LEXP}} \mathit{lval} : T \quad \tau,\gamma \vdash^{\text{EXP}} \mathit{exp} : T \quad (T \text{ static})}{\tau,\pi,\gamma \vdash^{\text{STMT}} \texttt{assign}\ \mathit{lval} = \mathit{exp} : C} \qquad \frac{\tau,\gamma \vdash^{\text{LEXP}} \mathit{lval} : D \quad \tau,\gamma \vdash^{\text{EXP}} \mathit{exp} : D}{\tau,\pi,\gamma \vdash^{\text{STMT}} \underline{\texttt{assign}}\ \mathit{lval} = \mathit{exp} : R}$$

$$\tau,\pi,\gamma \vdash^{\text{STMT}} \texttt{goto}\ m : C \qquad \tau,\pi,\gamma \vdash^{\text{STMT}} \underline{\texttt{goto}}\ m : R$$

$$\frac{\tau,\gamma \vdash^{\text{EXP}} \mathit{exp} : S}{\tau,\pi,\gamma \vdash^{\text{STMT}} \texttt{if}\ (\mathit{exp})\ m\ n : C} \qquad \frac{\tau,\gamma \vdash^{\text{EXP}} \mathit{exp} : D}{\tau,\pi,\gamma \vdash^{\text{STMT}} \underline{\texttt{if}}\ (\mathit{exp})\ m\ n : R}$$

$$\frac{\tau,\gamma \vdash^{\text{EXP}} \mathit{exp} : T \quad (T \text{ static})}{\tau,\pi \vdash^{\text{STMT}} \texttt{return}\ \mathit{exp} : C} \qquad \frac{\tau,\gamma \vdash^{\text{EXP}} \mathit{exp} : D}{\tau,\pi,\gamma \vdash^{\text{STMT}} \underline{\texttt{return}}\ \mathit{exp} : R}$$

$$\frac{\begin{array}{l}\tau,\gamma \vdash^{\text{EXP}} e_1 : T_1 \quad \ldots \quad \tau,\gamma \vdash^{\text{EXP}} e_n : T_n \\ \tau(f'\text{-}x_1) = T_1 \quad \ldots \quad \tau(f'\text{-}x_n) = T_n \\ \forall i = 1,\ldots,n : T_i \text{ static} \\ \pi(f') = T \quad \tau(x) = T \quad (T \text{ static})\end{array}}{\tau,\pi,\gamma \vdash^{\text{STMT}} \texttt{call}\ x = f'(e_1,\ldots,e_n) : C}$$

$$\frac{\begin{array}{l}\tau,\gamma \vdash^{\text{EXP}} e_1 : T_1 \quad \ldots \quad \tau,\gamma \vdash^{\text{EXP}} e_k : T_k \\ \tau(f'\text{-}x_1) = T_1 \quad \ldots \quad \tau(f'\text{-}x_k) = T_k \\ \forall i = 1,\ldots,k : T_i \text{ static} \\ \tau,\gamma \vdash^{\text{EXP}} e_{k+1} : D \quad \ldots \quad \tau,\gamma \vdash^{\text{EXP}} e_n : D \\ \tau(f'\text{-}x_{k+1}) = D \quad \ldots \quad \tau(f'\text{-}x_n) = D \\ \pi(f') = D \quad \tau(x) = D\end{array}}{\tau,\pi,\gamma \vdash^{\text{STMT}} \underline{\texttt{call}}\ x = f'(e_1,\ldots,e_n) : R}$$

$$\frac{\begin{array}{l}\tau,\gamma \vdash^{\text{EXP}} e_1 : T_1 \quad \ldots \quad \tau,\gamma \vdash^{\text{EXP}} e_n : T_n \\ \tau(f'\text{-}x_1) = T_1 \quad \ldots \quad \tau(f'\text{-}x_n) = T_n \\ \forall i = 1,\ldots,n : T_i \text{ static} \\ \pi(f') = T \quad \tau(x) = T \quad (T \text{ static})\end{array}}{\tau,\pi,\gamma \vdash^{\text{STMT}} \texttt{rcall}\ x = f'(e_1,\ldots,e_n) : C}$$

$$\frac{\begin{array}{l}\tau,\gamma \vdash^{\text{EXP}} e_1 : T_1 \quad \ldots \quad \tau,\gamma \vdash^{\text{EXP}} e_k : T_k \\ \tau(f'\text{-}x_1) = T_1 \quad \ldots \quad \tau(f'\text{-}x_k) = T_k \\ \forall i = 1,\ldots,k : T_i \text{ static} \\ \tau,\gamma \vdash^{\text{EXP}} e_{j+1} : D \quad \ldots \quad \tau,\gamma \vdash^{\text{EXP}} e_n : D \\ \tau(f'\text{-}x_{k+1}) = D \quad \ldots \quad \tau(f'\text{-}x_n) = D \\ \pi(f') = D \quad \tau(x) = D\end{array}}{\tau,\pi,\gamma \vdash^{\text{STMT}} \underline{\texttt{rcall}}\ x = f'(e_1,\ldots,e_n) : R}$$

Figure 16: Well-annotatedness rules for statements

This is not in agreement with the semantics, and it is cumberstone. Moreover, in order to get this to work, the dimension of the arrays must be known. With dynamically allocated arrays this will not always be the case. End of Example

Example 4.5 Recall the program from Chapter 3 (here in Core C):

```
int main()
{
    int a[10] int *p
    assign p = a
    assign *p = 87
}
```

It was claimed that `a` will not be split. Assume `a` is non-static. When the assignment `assign p = a` will require `a` to have type D, *i.e.* `a` will not be split.

Suppose the assignment is written `p = &a[0]`. When the rule for the address operator will *allow* the array to be split, and still the assignments `assign p = &a_0`, `assign p[0] = 87` will work as expected. The difference between `*p` and `p[`n`]` is that the first always assign the location it points to, but the latter form to variables not necessarily being allocated the same place in the residual program. End of Example

4.2.4 Well-Annotatedness for Functions

Definition 4.8 (Residual Function)
Let f be a function and τ a type assignment respecting the initial division. Then f is a residual function if

1. $\exists f\text{-}v_i : \tau(f\text{-}v_i) = D$ *(f has a dynamic parameter), or*
2. $\exists v \in \{v \in Id \mid v \text{ used in } f\} : \tau(v) = D$ *(f uses a dynamic global variable)*

where "use" means "appears in a lefthand or righthand side expression". □

Let the map $\mathcal{GF} : \mathrm{FId} \to \wp(\mathrm{Id})$ be defined for each function f to the set of global and call-by-reference passed variables (possibly) assigned in f. The map is obviously computable.

Definition 4.9 (Recursive Residual Function)
Let f be a residual function. If f is possibly direct or mutual recursive then f is a recursive residual function. □

A function which is neither residual nor recursive residual is called a *specialization time* function.

Definition 4.10 (Well-Annotatedness for Specialization time Functions)
Let p be a two-level program and f a function in p. Let $\mathcal{S}_f$ be the statements in f. Assume furthermore that a type assignment τ (respecting the initial division), a function type assignment π, and an operator type assignment γ is given. The function f is a well-annotated specialization time function if

1. $\forall i : \tau(f\text{-}x_i)$ *is static*
2. $\pi(f)$ *is static*
3. $\forall s \in \mathcal{S}_f : \tau, \pi, \gamma \vdash^{\mathrm{STMT}} s : C$

□

The first condition ensures that a specialization time function has no dynamic arguments. The second one assures that a static value is returned. The last condition requires all statements to be specialization time. Contained in the last condition is that a specialization time function neither use nor assign a non-local dynamic variable.

Definition 4.11 (Well-Annotatedness of Residual Functions)
Let a two-level Core C program p be given and let f a function in p with statements $\mathcal{S}_f$. Let τ be a type assignment (respecting the initial division), π a function type assignment and γ an operator type assignment. Function f is a well-annotated residual function if

1. $\forall i = 1, \ldots, n$: *if* $\tau(f\text{-}x_i)$ *not static then* $\tau(x_i) = D$
2. $\pi(f) = D$
3. *If f is recursive:*
   $\forall s \in \mathcal{S}_f, s = \texttt{assign}$ *lexp* `=` *exp* : *lexp* $\in \mathcal{GF}(f) \Rightarrow \tau, \pi, \gamma \vdash^{\text{STMT}} s : R$
4. $\forall s \in \mathcal{S}_f : \tau, \pi, \gamma \vdash s : U \in \{\mathcal{C}, \mathcal{R}\}$
5. $\forall s \in \mathcal{S}_f, s = \texttt{return}$ *exp* $: \tau, \pi, \gamma \vdash^{\text{STMT}} s : R$

where lexp $\in \mathcal{GF}(f)$ means the variable in lexp is in the set. □

The first condition captures that a residual function cannot receive partial static data structures. The second forces the return value of f to be dynamic. The third rule is for (possibly) recursive functions. In the affirmative case, all assignments to non-local variables are required to be dynamic. The fourth and fifth rules ensures well-annotatedness of the statements.

Definition 4.12 (Well-Annotatedness of Core C)
Let a Core C program p be given with functions $\mathcal{F}$, and goal parameters $x_1, \ldots, x_n$. Let τ be a type assignment, π a function type assignment and γ an operator type assignment. Further, let an initial division τ_0 be given and assume that τ respect τ_0. Program p is well-annotated iff

1. $\forall f \in \mathcal{F}$: *f is well-annotated*
2. *The main function f_0 is residual*

□

4.3 Two-Level Core C Semantics

We are now in a position to formally define the two-level semantics of two-level Core C. This also specify the C program specializer. The semantics consists of four parts: function specialization, poly-variant specialization of dynamic basic blocks, two-level execution of statements, and treatment of expressions. The section ends up with a definition of the meaning function $[\![\cdot]\!]_{2C}$.

For simplicity, liveness properties are not used in the semantics. This can easily be incorporated.

The general framework of this section is as follows. We assume given a well-annotateted two-level Core C program p, a specialization time store σ_0^S consistent with the division holding the values of the static input data, an environment, and an operator meaning function $\mathcal{O}$.

Furthermore, let the function *copy* : Store $\rightarrow$ Store be a specialization time copy function, as described in the previous chapter.

$$[\textit{sharing}\ ] \quad \frac{\mathcal{P}_{res}(f') = \langle f, \sigma^S, \mathcal{C}, \Sigma\rangle}{\rho \vdash^{\underline{\text{FUNC}}} \langle f, \sigma^S\rangle \Rightarrow \langle f', \Sigma\rangle}$$

$$[\textit{specialize}\ ] \quad \frac{f, \rho \vdash^{\underline{\text{BB}}} \langle l_{first}, \mathit{copy}(\sigma^S)\rangle \Rightarrow \langle \mathcal{C}, \Sigma\rangle \quad \mathcal{P}_{res}(f') = \langle f, \sigma^S, \mathcal{C}, \Sigma\rangle}{\rho \vdash^{\underline{\text{FUNC}}} \langle f, \sigma^S\rangle \Rightarrow \langle f', \Sigma\rangle}$$

Figure 17: Two-level semantics for function specialization

4.3.1 Two-level Semantics for Function Specialization

The specialization of a function is a transition relation from a function name and a specialization time store to a residual function name, the residual function and a set of end configuration stores:

$$\vdash^{\underline{\text{FUNC}}}: \text{FId} \times \text{Store} \rightarrow \text{FId} \times \text{Func} \times \wp(\mathbb{N} \times \text{Store}). \tag{4.3}$$

For reference purposes the end configuration stores are tagged with a number (corresponding to the number of the `endconf` variable).

For notational convenience we "globalize" the residual function code, and introduce a semantic function:

$$\mathcal{P}_{res} : \text{FId} \rightarrow \text{FId} \times \text{Store} \times \text{Code} \times \wp(\mathbb{N} \times \text{Store}) \tag{4.4}$$

similar to the $\mathcal{P}$ function. Here, an entry $\mathcal{P}_{res}(f') = \langle f, \sigma^S, \mathcal{C}, \Sigma\rangle$, means: f' is a specialized version of f with respect to store σ^S, with (residual) statements $\mathcal{C}$ and end configuration stores Σ. Hence, the new type of $\vdash^{\underline{\text{FUNC}}}$ is:

$$\vdash^{\underline{\text{FUNC}}}: \text{FId} \times \text{Store} \rightarrow \text{FId} \times \wp(\mathbb{N} \times \text{Store}). \tag{4.5}$$

Given $\mathcal{P}_{res}$, a residual program can be obtained by addition of variable declarations *etc.* We omit details.

Definition 4.13 (Two-Level semantics for Functions)
Let f be a function name and σ^S a store. Two-level execution of f yields residual function name f_{res} and the set of tagged end configuration stores Σ if

$$\rho \vdash^{\underline{\text{FUNC}}} \langle f, \sigma^S\rangle \Rightarrow \langle f_{res}, \Sigma\rangle$$

where $\vdash^{\underline{\text{FUNC}}}$ is defined in Figure 17. □

The sharing rule expresses that already specialized functions can be reused. The condition is that the function name and the specialization time store agrees. Otherwise the function must be specialized with respect to the static values in the specialization time store, and this is formalized via the $\vdash^{\underline{\text{BB}}}$ relation. Note how the signature of the residual function must be present in $\mathcal{P}_{res}$ (operational speaking: is *stored* in $\mathcal{P}_{res}$). The label l_{first} is the label of the first statement in function f. Before the specialization is initiated, the store is copied as discussed in the previous chapter.

$$[\textit{sharing}\ ] \quad \frac{\langle l, \sigma^S \rangle \in \mathcal{O}}{f, \rho \vdash^{\underline{\text{BB}}} \langle \{\langle l, \sigma^S \rangle\} \cup \mathcal{P}, \mathcal{O}, \mathcal{C}, \Sigma \rangle \Rightarrow \langle \mathcal{P}, \mathcal{O}, \mathcal{C}, \Sigma \rangle}$$

$$[\textit{specialize}\ ] \quad \frac{f, \rho \vdash^{\underline{\text{STMT}}} \langle l, \textit{copy}\sigma^S, [\,] \rangle \Rightarrow \langle \mathcal{P}', \mathcal{C}', \Sigma' \rangle}{f, \rho \vdash^{\underline{\text{BB}}} \langle \{\langle l, \sigma^S \rangle\} \cup \mathcal{P}, \mathcal{O}, \mathcal{C}, \Sigma \rangle \Rightarrow \langle \mathcal{P} \cup \mathcal{P}', \mathcal{O} \cup \{\langle l, \sigma^S \rangle\}, \mathcal{C} :: \mathcal{C}', \Sigma \cup \Sigma' \rangle}$$

$$[\textit{end}\ ] \quad f, \rho \vdash^{\underline{\text{BB}}} \langle \emptyset, \mathcal{O}, \mathcal{C}, \Sigma \rangle \Rightarrow \langle \mathcal{C}, \Sigma \rangle$$

$$[\textit{sequence}\ ] \quad \frac{\begin{array}{l} f, \rho \vdash^{\underline{\text{BB}}} \langle \mathcal{P}, \mathcal{O}, \mathcal{C}, \Sigma \rangle \Rightarrow \langle \mathcal{P}', \mathcal{O}', \mathcal{C}', \Sigma' \rangle \\ f, \rho \vdash^{\underline{\text{BB}}} \langle \mathcal{P}', \mathcal{O}', \mathcal{C}', \Sigma' \rangle \Rightarrow \langle \mathcal{P}'', \mathcal{O}'', \mathcal{C}'', \Sigma'' \rangle \end{array}}{f, \rho \vdash^{\underline{\text{BB}}} \langle \mathcal{P}, \mathcal{O}, \mathcal{C}, \Sigma \rangle \Rightarrow \langle \mathcal{P}'', \mathcal{O}'', \mathcal{C}'', \Sigma'' \rangle}$$

$$[\textit{sequence}\ ] \quad \frac{\begin{array}{l} f, \rho \vdash^{\underline{\text{BB}}} \langle \mathcal{P}, \mathcal{O}, \mathcal{C}, \Sigma \rangle \Rightarrow \langle \mathcal{P}', \mathcal{O}', \mathcal{C}', \Sigma' \rangle \\ f, \rho \vdash^{\underline{\text{BB}}} \langle \mathcal{P}', \mathcal{O}', \mathcal{C}', \Sigma' \rangle \Rightarrow \langle \mathcal{C}'', \Sigma'' \rangle \end{array}}{f, \rho \vdash^{\underline{\text{BB}}} \langle \mathcal{P}, \mathcal{O}, \mathcal{C}, \Sigma \rangle \Rightarrow \langle \mathcal{C}'', \Sigma'' \rangle}$$

Figure 18: Two-level operational semantics dynamic basic blocks

4.3.2 Two-level Semantics for Dynamic Basic Blocks

The two-level semantics for dynamic basic blocks formalizes the specialization of a function body. The semantics is specified in terms of the transition relation $\vdash^{\underline{\text{BB}}}$. It is a transition from a set of specialization points ("poly-set"), another set of specialization points ("seen before"), a code list, and a set of (tagged) *end configuration* stores. The final result is the generated code and the set of (tagged) end configuration stores.

$$\begin{array}{rll} \vdash^{\underline{\text{BB}}}\!\!: & \wp(\text{Label} \times \text{Store}) \times \wp(\text{Label} \times \text{Store}) & \times \text{Code} \times \wp(\mathbb{N} \times \text{Store}) \\ & & \rightarrow \text{Code} \times \wp(\mathbb{N} \times \text{Store}) \end{array} \tag{4.6}$$

The two first arguments are the *pending* and the *seen before* (out) sets. The third is the *code* argument and the fourth the set of tagged end configuration stores.

Definition 4.14 (Two Level Semantics for Basic Blocks)
Let l be a specialization point and σ^S a specialization time store. Two-level execution of the basic blocks starting at label l yields the code list $\mathcal{C}$ and the set of end configuration stores Σ if

$$f, \rho \vdash^{\underline{\text{BB}}} \langle \{\langle l, \sigma^S \rangle\}, \emptyset, [\,], \emptyset \rangle \Rightarrow \langle \mathcal{C}, \Sigma \rangle$$

where the transition relation $\vdash^{\underline{\text{BB}}}$ is defined in Figure 18. □

We use the symbol :: to denote "concatenation" of statement lists. The symbol [] denotes an empty code list. The intended meaning should be clear.

The two rules sharing and specialize does the real work. If the first residual program point in pending exist in out, the residual code can be shared (sharing). Otherwise the

dynamic basic block must be specialized yielding a list of residual statements and possibly some new residual program points to be processed.

Note, liveness information can be incorporated in the semantics by filtering out dead variables when specialization time stores are compared in the sharing rule.

The processing of dynamic basic blocks ends when the pending list is empty. The sequence rules makes to game to run. In an implementation, finiteness of pending must of course be assured (by looking at the out-set before inserting new elements).

4.3.3 Two-level Operational Semantics for Statements

The two-level semantics for statements formalize execution of static statements and code generation for dynamic statements. Recall that the specialization of dynamic basic blocks result in some code, a set of new specialization points, and possibly an end configuration store. The dynamic basic block is identified by a *specialization* point, *i.e.* a label. The two-level semantics for statements is a transition relation of type:

$$\vdash^{\underline{\text{STMT}}}: \text{Label} \times \text{Store} \times \text{Code} \rightarrow \wp(\text{Label} \times \text{Store}) \times \text{Code} \times \wp(I\!N \times \text{Store}), \qquad (4.7)$$

where the result is a set of residual programp points, a code list and a set of end configuration stores.

New specialization points are generated due to two-level execution of dynamic `goto`, `if`, or `call` statements. An end configuration store is returned if the dynamic basic blocks end with a `return`. Two-level execution takes place in the context of a function name and an enviroment.

Definition 4.15 (Two-level Execution of Statements)
Let f be a function name and B a dynamic basic block identified by label l, and let σ^S be a specialization time store. Two-level execution of B yields the specialization set $\mathcal{P}$, the code list $\mathcal{C}$, and end configuration set Σ if

$$f, \rho \vdash^{\underline{\text{STMT}}} \langle l, \sigma^S, [\,] \rangle \Rightarrow \langle \mathcal{P}, \mathcal{C}, \Sigma \rangle$$

where the relation $\vdash^{\underline{\text{STMT}}}$ is defined in Figure 19. □

The semantic function *update* takes care of the updating of the store after a residual `call` statement. Its formal definition is given below.

Definition 4.16 (The Update Operator)
The operator update : FId × Store × $\wp$(Store) → $\wp$(Store) is defined by

$$update(f, \sigma, \Sigma) = \{upd(f, \sigma, \sigma'))|\sigma' \in \Sigma\}.$$

The operator upd : FId × Store × Store → Store is defined by

$$upd(f, \sigma, \sigma') = \sigma''$$

where for all locations $l \in dom(\sigma)$:

$$\sigma''(l) = \begin{cases} \sigma(l') & \text{if } x_i \text{ is call-by-value parameter} \\ \sigma'(l) & \text{if } x_i \text{ is call-by-reference parameter} \\ \sigma'(l) & \text{if } x_i \text{ is global variable} \end{cases}$$

with x_i ranging over global variables and the parameters of f, $\rho, \sigma \vdash^{\text{LVAL}} x_i \Rightarrow l$ and $\rho, \sigma' \vdash^{\text{LVAL}} x_i \Rightarrow l'$ for all i. □

4.3.4 Two-level Operational Semantics for Expressions

The reduction of an expression is a transition relation from a two-level expression to some code. The specialization time store is needed in case the expression contains a static subexpression.

$$\vdash^{\underline{\text{EXP}}}: \text{Expr} \rightarrow \text{Expr} \tag{4.8}$$

Definition 4.17 (Reduction of expressions)
Let e be an expression and σ^S a compile-time store. Reduction of e yields the expression e' if

$$\rho, \sigma^S \vdash^{\underline{\text{EXP}}} e \Rightarrow e'$$

using the transition rules as defined in Figure 21. □

The rules applies only for dynamic expressions. However, there are two rules for `index` and `struct` expressions. The non-underlined are the rules for splitting of arrays and structures.

Definition 4.18 (Reduction of left-expressions)
Let e be a left-expression and σ^S a compile-time store. Reduction of e is the expression e' iff

$$\sigma^S \vdash^{\underline{\text{LVAL}}} e \Rightarrow e'$$

where the relation $\vdash^{\underline{\text{LVAL}}}$ is defined in Figure 22. □

4.3.5 Two-level Operational Semantics for Core C

Definition 4.19 (Two-level Operational Semantics of Core C)
Let a well-annotated two-level Core C program p with main function f_0 be given, and assume a store σ^S representing the static global variables and static input $v_1, \ldots, v_k$. Let ρ be an environment consistent with σ^S. Then

$$[\![p]\!]_{2\mathcal{C}}(v_1, \ldots, v_k) \Rightarrow p'$$

iff the program variable $\mathcal{P}$ represent p and

$$\rho \vdash^{\underline{\text{FUNC}}} \langle f_0, \sigma^S \rangle \Rightarrow \langle f', \Sigma \rangle$$

and $\mathcal{P}_{res}$ represent the program p'. □

The function f' is the goal function in the residual program.

$$[assign\ ]\quad \frac{\mathcal{P}(f)(l) = \texttt{l: assign } lval \texttt{ = } exp \quad \rho, \sigma^S \vdash^{\text{LVAL}} lval \Rightarrow loc \quad \rho, \sigma^S \vdash^{\text{EXP}} exp \Rightarrow v}{f, \rho \vdash^{\underline{\text{STMT}}} \langle l, \sigma^S, \mathcal{C}\rangle \Rightarrow \langle l', \sigma^S[v/loc], \mathcal{C}\rangle}$$

$$[\underline{assign}\ ]\quad \frac{\mathcal{P}(f)(l) = \texttt{l: } \underline{\texttt{assign}} \ lval \texttt{ = } exp \quad \rho, \sigma^S \vdash^{\underline{\text{LVAL}}} lval \Rightarrow lval' \quad \rho, \sigma^S \vdash^{\underline{\text{EXP}}} exp \Rightarrow exp'}{f, \rho \vdash^{\underline{\text{STMT}}} \langle l, \sigma^S, \mathcal{C}\rangle \Rightarrow \langle l', \sigma^S, \mathcal{C} :: [\langle l, \sigma^S\rangle : \texttt{ assign } lval' = exp']\rangle}$$

$$[goto\ ]\quad \frac{\mathcal{P}(f)(l) = \texttt{l: goto } m}{f, \rho \vdash^{\underline{\text{STMT}}} \langle l, \sigma^S, \mathcal{C}\rangle \Rightarrow \langle m, \sigma^S, \mathcal{C}\rangle}$$

$$[if\ ]\quad \frac{\mathcal{P}(f)(l) = \texttt{l: if (}exp\texttt{)} \ m \ n \quad \rho, \sigma^S \vdash^{\text{EXP}} exp \Rightarrow v \quad v \downarrow \text{Int} \neq 0}{f, \rho \vdash^{\underline{\text{STMT}}} \langle l, \sigma^S, \mathcal{C}\rangle \Rightarrow \langle m, \sigma^S, \mathcal{C}\rangle}$$

$$[if\ ]\quad \frac{\mathcal{P}(f)(l) = \texttt{l: if (}exp\texttt{)} \ m \ n \quad \rho, \sigma^S \vdash^{\text{EXP}} exp \Rightarrow v \quad v \downarrow \text{Int} = 0}{f, \rho \vdash^{\underline{\text{STMT}}} \langle l, \sigma^S, \mathcal{C}\rangle \Rightarrow \langle n, \sigma^S, \mathcal{C}\rangle}$$

$$[call\ ]\quad \frac{\begin{array}{l}\mathcal{P}(f)(l) = \texttt{l: call } x \texttt{ = } f'\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} \\ \rho, \sigma^S \vdash^{\text{EXP}} e_1 \Rightarrow v_1 \ldots \rho, \sigma^S \vdash^{\text{EXP}} e_m \Rightarrow v_m \\ \rho \circ [x_1 \mapsto l_1, \ldots, x_n \mapsto l_n] \vdash^{\text{FUNC}} \\ \langle f', \sigma^S[v_1/l_1, \ldots, v_n/l_n]\rangle \Rightarrow \langle v, \sigma^{S'}[\cdot/l_1, \ldots, \cdot/l_n]\rangle\end{array}}{f, \rho \vdash^{\underline{\text{STMT}}} \langle l, \sigma^S, \mathcal{C}\rangle \Rightarrow \langle l', \sigma^{S'}[v/\rho(x)], \mathcal{C}\rangle}$$

$$[rcall\ ]\quad \frac{\begin{array}{l}\mathcal{P}(f)(l) = \texttt{l: rcall } x \texttt{ = } f'\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} \\ \rho, \sigma^S \vdash^{\text{EXP}} e_1 \Rightarrow v_1 \ldots \rho, \sigma^S \vdash^{\text{EXP}} e_m \Rightarrow v_m \\ \rho \circ [x_1 \mapsto l_1, \ldots, x_n \mapsto l_n] \vdash^{\text{FUNC}} \\ \langle f', \sigma^S[v_1/l_1, \ldots, v_n/l_n]\rangle \Rightarrow \langle v, \sigma^{S'}[\cdot/l_1, \ldots, \cdot/l_n]\rangle\end{array}}{f, \rho \vdash^{\underline{\text{STMT}}} \langle l, \sigma^S, \mathcal{C}\rangle \Rightarrow \langle l', \sigma^{S'}[v/\rho(x)], \mathcal{C}\rangle}$$

$$[\underline{rcall}\ ]\quad \frac{\begin{array}{l}\mathcal{P}(f)(l) = \texttt{l: } \underline{\texttt{rcall}} \ x \texttt{ = } f'\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} \\ \rho, \sigma^S \vdash^{\text{EXP}} e_1 \Rightarrow v_1 \ldots \rho, \sigma^S \vdash^{\text{EXP}} e_k \Rightarrow v_k \\ \rho, \sigma^S \vdash^{\underline{\text{EXP}}} e_{k+1} \Rightarrow e'_{j+1} \ldots \rho, \sigma^S \vdash^{\underline{\text{EXP}}} e_n \Rightarrow e'_n \\ \rho \circ [x_1 \mapsto l_1, \ldots, x_k \mapsto l_k] \vdash^{\underline{\text{FUNC}}} \langle f', \sigma^S[v_1/l_1, \ldots, v_k/x_k\rangle \Rightarrow \langle f_{res}, \emptyset\rangle\end{array}}{f, \rho \vdash^{\underline{\text{STMT}}} \langle l, \sigma^S, \mathcal{C}\rangle \Rightarrow \langle l', \sigma^S, \mathcal{C} :: \langle l, \sigma^S\rangle : \text{ call } x = f_{res}(e'_{j+1}, \ldots, e'_n)\rangle}$$

Figure 19: Two-level operational semantics for statements

$$[\underline{goto}\,] \quad \frac{\mathcal{P}(f)(l) = \underline{\texttt{goto}}\ m}{\begin{array}{l} f, \rho \vdash^{\underline{\text{STMT}}} \langle l, \sigma^S, \mathcal{C}\rangle \Rightarrow \\ \langle \{\langle m, \sigma^S\rangle\}, \mathcal{C} :: [\langle l, \sigma^S\rangle : \texttt{goto}\ \langle m, \sigma^S\rangle], \emptyset\rangle \end{array}}$$

$$[\underline{if}\,] \quad \frac{\begin{array}{l} \mathcal{P}(f)(l) = \underline{\texttt{if}}\ \texttt{(}exp\texttt{)}\ m\ n \\ \rho, \sigma^S \vdash^{\underline{\text{EXP}}} exp \Rightarrow exp' \end{array}}{\begin{array}{l} f, \rho \vdash^{\underline{\text{STMT}}} \langle l, \sigma^S, \mathcal{C}\rangle \Rightarrow \\ \quad \langle \{\langle m, \sigma^S\rangle, \langle n, \sigma^S\rangle\}, \mathcal{C} :: \langle l, \sigma^S\rangle : \texttt{if}\ (exp')\langle m, \sigma^S\rangle\ \langle n, \sigma^S\rangle, \emptyset\rangle \end{array}}$$

$$[\underline{call}\,] \quad \frac{\begin{array}{l} \mathcal{P}(f)(l) = \underline{\texttt{call}}\ x\ \texttt{=}\ f'\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} \\ \rho, \sigma^S \vdash^{\text{EXP}} e_1 \Rightarrow v_1 \cdots \rho, \sigma^S \vdash^{\text{EXP}} e_k \Rightarrow v_k \\ \rho, \sigma^S \vdash^{\underline{\text{EXP}}} e_{k+1} \Rightarrow e'_{k+1} \cdots \rho, \sigma^S \vdash^{\underline{\text{EXP}}} e_n \Rightarrow e'_n \\ \rho \circ [x_1 \mapsto l_1, \ldots, x_k \mapsto l_k] \vdash^{\underline{\text{FUNC}}} \\ \quad \langle f', \sigma^S[v_1/l_l, \ldots, v_k/l_k]\rangle \Rightarrow \langle f_{res}, \{\langle 1, \sigma_1\rangle, \ldots, \langle m, \sigma_m\rangle\}\rangle \\ update(f', \sigma^S, \{\sigma_1^S, \ldots, \sigma_m^S\}) = \{\sigma_1^{S\prime}, \ldots, \sigma_m^{S\prime}\} \end{array}}{\begin{array}{l} f, \rho \vdash^{\underline{\text{STMT}}} \langle l, \sigma^S, \mathcal{C}\rangle \Rightarrow \\ \quad \langle \{\langle l_1, \sigma_1^{S\prime}\rangle, \ldots, \langle l_m, \sigma_m^{S\prime}\rangle\}, \\ \qquad \mathcal{C} \ :: [\langle l, \sigma^S\rangle : \texttt{call}\ x = f_{res}(e'_1, \ldots, e'_n)] \\ \qquad\quad :: [l_1 : \texttt{if (endconf == 1)}\ \langle l_1, \sigma_1^{S\prime}\rangle\ \langle l_2, \sigma_2^{S\prime}\rangle] \\ \qquad\quad \ldots \\ \qquad\quad :: [l_m : \texttt{if (endconf ==}\ m\texttt{)}\ \langle l_m, \sigma_m^{S\prime}\rangle\ \texttt{error}], \\ \qquad \emptyset\rangle \end{array}}$$

$$[\underline{return}\,] \quad \frac{\begin{array}{l} \mathcal{P}(f)(l) = \underline{\texttt{return}}\ exp \\ \rho, \sigma^S \vdash^{\underline{\text{EXP}}} exp \Rightarrow exp' \\ i \text{ is fresh number} \end{array}}{\begin{array}{l} f, \rho \vdash^{\underline{\text{STMT}}} \langle l, \sigma^S, \mathcal{C}\rangle \Rightarrow \\ \quad \langle \emptyset, \\ \qquad \mathcal{C} \ :: [\langle l, \sigma^S\rangle : \texttt{assign endconf} = i] \\ \qquad\quad :: [\langle l', \sigma^S\rangle : \texttt{return}\ exp'], \\ \quad \{\langle i, \sigma^S\rangle\}\rangle \end{array}}$$

$$[sequence\,] \quad \frac{f, \rho \vdash^{\underline{\text{STMT}}} \langle l, \sigma^S, \mathcal{C}\rangle \Rightarrow \langle m, \sigma^{S\prime}, \mathcal{C}'\rangle \quad f, \rho \vdash^{\underline{\text{STMT}}} \langle m, \sigma^{S\prime}, \mathcal{C}'\rangle \Rightarrow \langle n, \sigma^{S\prime\prime}, \mathcal{C}''\rangle}{f, \rho \vdash^{\underline{\text{STMT}}} \langle l, \sigma^S, \mathcal{C}\rangle \Rightarrow \langle n, \sigma^{S\prime\prime}, \mathcal{C}''\rangle}$$

$$[sequence\,] \quad \frac{f, \rho \vdash^{\underline{\text{STMT}}} \langle l, \sigma^S, \mathcal{C}\rangle \Rightarrow \langle m, \sigma^{S\prime}, \mathcal{C}'\rangle \quad f, \rho \vdash^{\underline{\text{STMT}}} \langle m, \sigma^{S\prime}, \mathcal{C}'\rangle \Rightarrow \langle \mathcal{C}'', \Sigma\rangle}{f, \rho \vdash^{\underline{\text{STMT}}} \langle l, \sigma^S, \mathcal{C}\rangle \Rightarrow \langle \mathcal{C}'', \Sigma\rangle}$$

Figure 20: Two-level operational semantics for statements (cont)

$$[\underline{lift}\,] \quad \frac{\rho, \sigma^S \vdash^{\underline{\text{EXP}}} e \Rightarrow v}{\rho, \sigma^S \vdash^{\underline{\text{EXP}}} \underline{\texttt{lift}}\ e \Rightarrow \texttt{cst}\ v}$$

$$[\underline{var}\,] \quad \rho, \sigma^S \vdash^{\underline{\text{EXP}}} \underline{\texttt{var}}\ v \Rightarrow \texttt{var}\ v$$

$$[index\,] \quad \frac{\rho, \sigma^S \vdash^{\underline{\text{EXP}}} e_1 \Rightarrow \texttt{var}\ v \quad \rho, \sigma^S \vdash^{\text{EXP}} e_2 \Rightarrow i}{\rho, \sigma^S \vdash^{\underline{\text{EXP}}} \texttt{index}\ e_1\texttt{[}e_2\texttt{]} \Rightarrow \texttt{var}\ v\_i}$$

$$[\underline{index}\,] \quad \frac{\rho, \sigma^S \vdash^{\underline{\text{EXP}}} e_1 \Rightarrow e_1' \quad \rho, \sigma^S \vdash^{\underline{\text{EXP}}} e_2 \Rightarrow e_2'}{\rho, \sigma^S \vdash^{\underline{\text{EXP}}} \underline{\texttt{index}}\ e_1[e_2] \Rightarrow \texttt{index}\ e_1'\texttt{[}e_2'\texttt{]}}$$

$$[struct\,] \quad \frac{\rho, \sigma^S \vdash^{\underline{\text{EXP}}} e \Rightarrow \texttt{var}\ v}{\rho, \sigma^S \vdash^{\underline{\text{EXP}}} \text{struct}\ e.y \Rightarrow \texttt{var}\ v\_y}$$

$$[\underline{struct}\,] \quad \frac{\rho, \sigma^S \vdash^{\underline{\text{EXP}}} e \Rightarrow e'}{\rho, \sigma^S \vdash^{\underline{\text{EXP}}} \underline{\texttt{struct}}\ e.y \Rightarrow \texttt{struct}\ e'.y}$$

$$[\underline{uop}\,] \quad \frac{\rho, \sigma^S \vdash^{\underline{\text{EXP}}} e \Rightarrow e'}{\rho, \sigma^S \vdash^{\underline{\text{EXP}}} \underline{\texttt{uop}}\ op\ e \Rightarrow \texttt{uop}\ \ op\ \ e'}$$

$$[\underline{bop}\,] \quad \frac{\rho, \sigma^S \vdash^{\underline{\text{EXP}}} e_1 \Rightarrow e_1' \quad \rho, \sigma^S \vdash^{\underline{\text{EXP}}} e_2 \Rightarrow e_2'}{\sigma^S \vdash^{\underline{\text{EXP}}} \underline{\texttt{bop}}\ op\ e_1\ e_2 \Rightarrow \texttt{bop}\ \ op\ e_1'\ e_2'}$$

$$[\underline{ecall}\,] \quad \frac{\rho, \sigma^S \vdash^{\underline{\text{EXP}}} e_1 \Rightarrow e_1' \quad \ldots \quad \rho, \sigma^S \vdash^{\underline{\text{EXP}}} e_n \Rightarrow e_n'}{\rho, \sigma^S \vdash^{\underline{\text{EXP}}} \underline{\texttt{ecall}}\ f(e_1,\ldots,e_n) \Rightarrow \texttt{ecall}\ \ f\texttt{(}e_1'\texttt{,}\ldots\texttt{,}e_n'\texttt{)}}$$

Figure 21: Reduction of two-level expressions

$$[\underline{var}\,] \quad \tau, \gamma \vdash^{\underline{\text{LVAL}}} \underline{\texttt{var}}\ v \Rightarrow \texttt{var}\ v$$

$$[index\,] \quad \frac{\sigma^S \vdash^{\underline{\text{LVAL}}} e_1 \Rightarrow \texttt{var}\ v \quad \sigma^S \vdash^{\text{EXP}} e_2 \Rightarrow i}{\sigma^S \vdash^{\underline{\text{LVAL}}} \texttt{index}\ e_1[e_2] \Rightarrow \texttt{var}\ v\_i}$$

$$[\underline{index}\,] \quad \frac{\sigma^S \vdash^{\underline{\text{LVAL}}} e_1 \Rightarrow e_1' \quad \sigma^S \vdash^{\underline{\text{EXP}}} e_2 \Rightarrow e_2'}{\sigma^S \vdash^{\underline{\text{LVAL}}} \underline{\texttt{index}}\ e_1[e_2] \Rightarrow \texttt{index}\ e_1'[e_2']}$$

Figure 22: Two-level operational semantics for left-expressions

4.4 Correctness

In order to show the correctness of the two-level semantics, we must prove the validity of the *Mix Equation*.

Let p be a program and d_1, d_2 be input values. Then it holds

$$[\![p^{ann}]\!]_{2C} d_1 \Rightarrow p' \quad \text{and} \quad [\![p']\!]_C d_2 \Rightarrow d$$

only if

$$[\![p]\!]_C (d_1, d_2) \Rightarrow d,$$

where p^{ann} is a well-annotated two-level program for p corresponding to the initial division d_1 static and d_2 dynamic.

Since the specialization may be non-terminating, we cannot show "if and only if".

The proof would be in two steps. First it must be shown that if a program is well-annotated, then the semantics give the program a meaning (unless it is non-terminating). This correspond to showing that there "exist" a rules for every language construct and that it is well-formed. The second part of the proof would be a induction proof over the syntax formally showing that the correct code is generated.

A complete correctness proof would be hard (and lengthy!) due to the complexity of both the language and the techniques applied. A complete correctness proof for a partial evaluator for the untyped lambda calculus is given in [Gomard 1992].

4.5 Summary

In this chapter we have introduced two-level Core C. We defined its syntax, and expressed *well-annotatedness* rules which guarantees that specialization of a *well-annotated* two-level Core C program goes "well". Finally we precisely defined a two-level Core C operational semantics. An implementation of the semantics correspond to a specializer for Core C.

Chapter 5

Binding Time Analysis

This chapter contains the description of a fully automatic binding time analysis for Core C. Recall the purpose of the binding time analysis: given a Core C program and an initial input division, to compute a *congruent* division of all the variables in the program. The division must fulfill the *well-annotatedness* requirements defined in Chapter 4. Given the division, a *well-annotated* Core C program can easily be derived.

Traditionally, binding time analysis has been done by abstract interpretation over, for instance, the two-point domain $S \sqsubset D$ [Jones *et al.* 1989]. Later, binding time analyses using more informative domains have been developed, for example, binding time analysis for a higher order subset of Scheme [Bondorf 1990a] [Consel 1990], and analyses treating partially static data structures [Mogensen 1988]. The binding time analysis for Core C also include partially static data structures.

More recently, the Nielsons have formulated the binding time analysis problem as a *type inference* problem using *types* to describe the binding time of expressions. The analysis was for a two-level typed version of the lambda calculus [Nielson and Nielson 1988b]. Gomard devised a binding time analysis for a two-level, untyped lambda calculus and obtained an algorithm using a backtracking version of the type inference algorithm W [Gomard 1990]. The algorithm required, however, user insertion of lift operators. Andersen and Mossin extended the method with coercions to deal with the lift problem, and obtained a binding time analysis for a higher order, side-effect free subset of Scheme [Andersen and Mossin 1990].

Henglein reformulated the problem of doing binding time analysis for the untyped lambda calculus into the solution of a *constraint set*, and gave an efficient, almost linear time algorithm [Henglein 1991b].

The analysis presented in this chapter is an adaption Henglein's method. Due to the complexity and the differences in the languages, new aspects are added, though. Knowledge to Henglein's binding time analysis [Henglein 1991b] is in general expected in this chapter.

The chapter is organized as follows. First we give some preliminary definitions. We define two maps giving information about the use of non-local variables and recursive functions. The definition of binding time types is also recalled.

Section 5.2 contains the definitions of *constraints* and *constraint systems*. We define

$$
\begin{array}{l}
\mathcal{RS} : \mathsf{Stmt} \to \wp(\mathsf{Fld}) \\
\quad \mathcal{RS}[\![l : \texttt{assign } \mathit{lval} \texttt{ = } \mathit{exp}]\!]\phi \quad = \emptyset \\
\quad \mathcal{RS}[\![l : \texttt{goto } m]\!]\phi \quad = \emptyset \\
\quad \mathcal{RS}[\![l : \texttt{if (}\mathit{exp}\texttt{) } m \ n]\!]\phi \quad = \emptyset \\
\quad \mathcal{RS}[\![l : \texttt{return } \mathit{exp}]\!]\phi \quad = \emptyset \\
\quad \mathcal{RS}[\![l : \texttt{call } x \texttt{ = } f\texttt{(}e_1\texttt{, ..., }e_n\texttt{)}]\!]\phi \quad = \{f\} \cup \phi(f)
\end{array}
$$

Figure 23: Definition of the map $\mathcal{RS}$

solution to and *normal form* of constraint systems.

The binding time algorithm is given in Section 5.3. The algorithm is a slightly modified version of Henglein's, and we therefore omit details and refer to his paper. Section 5.4 contains an example of the analysis applied to a program. The last section discuss *termination* and *finite divisions*.

5.1 Preliminary Definitions

For doing binding time analysis we need information about the use of non-local variables and possibly recursive functions. We define two maps $\mathcal{GF}$ and $\mathcal{RF}$ expressing this. Both maps are undecidable and hence only (safe) approximations can be computed.

Definition 5.1 (Non-local Variable Map)
Let a Core C program be given. For all functions f define the map $\mathcal{GF} : FId \to \wp(Id)$ by

$$\mathcal{GF}(f) = \{s \in \mathcal{S}_f \mid s = \texttt{assign } \mathit{lval} \texttt{ = } \mathit{exp} \wedge v \in \mathit{lval} \wedge v \in \mathcal{NL}\}$$

where $\mathcal{NL}$ is the set of global variable and call-by-reference variables of f. □

The map $\mathcal{GF}$ map was used in Chapter 4.

Definition 5.2 (Recursive Functions) ‘
Let a Core C program be given. For all functions f, define $\mathcal{RF} : FId \to \wp(FId)$ by

$$\mathcal{RF}(f) = \mathit{Fix}\ \lambda\phi.\ \bigcup_{s \in \mathcal{S}_f} \mathcal{RS}[\![s]\!]\phi$$

where the definition of $\mathcal{RS}$ is given in Figure 23.

□

Informally, for a function f, $\mathcal{RF}(f)$ equals the set of function names which f *may* call. Especially, if $f \in \mathcal{RF}(f)$ when f is a possibly recursive function.

Recall that the types describing the binding time of expressions were defined in Section 4.2 as follows:

$$\mathcal{T} ::= S \mid D \mid [\mathcal{T}] \mid \mathcal{T} \otimes \mathcal{T} \tag{5.1}$$

where S and D denotes *static* and *dynamic* respectively, $[T]$ describes a statically indexed array with elements of type T, and the product is for structures. In the rest of this chapter we use the letter T to range over binding time types (a "*type variable*").

We impose two irreflexive partial orders on the binding time types as follows.

Definition 5.3 (Order on Binding Types)
Define the irreflexive partial orders $\prec$ and $\sqsubset$ by:

(Lift) $S \prec D$

(Split) $[D] \sqsubset D$

(Split) $\underbrace{D \otimes \cdots \otimes D}_{n} \sqsubset D$

for all $n > 1$. No other binding time types are related. □

The Lift rule expresses that a static (base) value can be lifted using the lift operator `@` yielding a dynamic value. The Split rule contains the splitting of arrays and structures. A statically indexed array with dynamic elements can be split into separate variables yielding something of binding time type D. Similarly for a structure with dynamic fields.

Let a Core C program p be given. We define a *completion* of p to be a *well-annotated* two-level Core C program such that if all the underlining and lifts are removed, it equals p. Define the order $\leq$ on binding time types by $S \leq D$, $[T_1] \leq [T_2] \Leftrightarrow T_1 \leq T_2$, and $T_1 \otimes T_2 \leq T_3 \otimes T_4 \Leftrightarrow T_1 \leq T_3$ and $T_2 \leq T_4$ on binding time types, and lift it pointwise to type assignments. Using the order $\leq$ we define a *minimal completion* the obvious way. The binding time analysis should preferable compute a minimal completion to the given program.

5.2 Constraint Systems

The complete binding time analysis consists of four parts: construction of a *constraint set*, *normalization* of the constraint set, determination of a *minimal solution*, and *annotation* of the program. In this section we describe the construction of a constraint set which exactly captures the binding time requirements of a program.

The constraint set is mainly constructed on basis of the expressions, but statements and functions imposes some extra conditions. For example, in an assignment statement, if one of the expressions on the lefthand or righthand side are non-static, then both must be D, cf. the well-annotatedness rules for statements, Definition 4.7.

5.2.1 Constraints

Definition 5.4 (Constraint System)
A constraint system is a (multi)set[1] *of constraints of the following form:*

$$T_1 \preceq T_2 \qquad T_1 \sqsubseteq T_2 \qquad T_1 = T_2 \qquad T_1 \rhd T_2 \qquad T_1 \bowtie T_2$$

where T_1, T_2 are type variables ranging over binding time types. □

[1] A multi set is an improper set where an element may occur several times

A solution to a constraint set is a substitution from type variables to binding time types.

DEFINITION 5.5 (SOLUTION TO CONSTRAINT SET)
A substitution $\Theta : TypeVar \rightarrow \mathcal{T}$ is a solution to a constraint set $\mathcal{C}$ iff

1. *For all $T_1 \preceq T_2 \in \mathcal{C}$: $\Theta(T_1) \preceq \Theta(T_2)$*
2. *For all $T_1 \sqsubseteq T_2 \in \mathcal{C}$: $\Theta(T_1) \sqsubseteq \Theta(T_2)$*
3. *For all $T_1 = T_2 \in \mathcal{C}$: $\Theta(T_1) = \Theta(T_2)$*
4. *For all $T_1 \rhd T_2 \in \mathcal{C}$: if $\Theta(T_1) = D$ then $\Theta(T_2) = D$*
5. *For all $T_1 \bowtie T_2 \in \mathcal{C}$: if D occurs in $\Theta(T_1)$ then $\Theta(T_2) = D$*
6. *For all other type variables T_1 not in $\mathcal{C}$: $\Theta(T_1) = T_1$*

where T_1, T_2 are meta variables over binding time types. The set of solutions to $\mathcal{C}$ is also written Sol($\mathcal{C}$). □

OBSERVATION 5.1 *The constraint $T_1 \rhd T_2$ expresses a dependency: If T_1 is dynamic then T_2 is dynamic. The constraint $T_1 \bowtie T_2$ expresses a strong dependency: If T_1 is non-static, i.e. D occurs in T_1, then T_2 is dynamic.* END OF OBSERVATION

The strong dependency constraint is not needed when doing binding time analysis in the lambda calculus. It is needed here mainly due to *partially static data structures*.

5.2.2 Constraint Set for Expressions

Let a program be given. To each expression e appearing in it we assign two unique type variables T_e and $\overline{T}_e$. To every variable v (global, parameter or local variable) we assign a unique variable T_v. For simplicity we assume as usual all variable names to be different.

Intuitively, for an expression e, the type variable T_e is the *normal* binding time type of e and $\overline{T}_e$ the *lifted* binding time type, *i.e.* the binding time type obtained if the lift operator is applied. When all the type variables have been instantiated and an expression e has normal binding time S but lifted binding time D, a lift operator must be inserted in order to well-annotate the program.

For each function f we assign a type variable $T_{\pi(f)}$ which describes the binding time type of the value f returns. This is in agreement with the mono-variant binding time assignment to functions.

DEFINITION 5.6 (CONSTRAINT SET FOR EXPRESSIONS)
Let e be a Core C expression. The constraint set $\mathcal{C}_{exp}(e)$ for e is defined inductively as shown in Figure 24. □

For all expressions e, the constraint $T_e \preceq \overline{T}_e$ is included. It says that the expression e (in some cases) can be lifted to a dynamic value. The cases where the lift operator *cannot* be applied are ruled out by extra constraints.

$\mathcal{C}_{exp}(e)$ = *case* e *of*

`cst` c	$: \{S = T_e, T_e \preceq \overline{T}_e\}$
`var` v	$: \{T_v = T_e, T_e \preceq \overline{T}_e\}$
`index` e_1`[`e_2`]`	$: \{[T_e] \sqsubseteq \overline{T}_{e_1}, T_{e_1} = \overline{T}_{e_1}, T_{e_1} \rhd \overline{T}_{e_2}, T_{e_2} \rhd \overline{T}_{e_1}, T_e \preceq \overline{T}_e\}$ $\cup\, \mathcal{C}(e_1) \cup \mathcal{C}(e_2)$
`struct` e_1`.`y	$: \{T_1 \otimes \cdots \otimes T_e \otimes \cdots \otimes T_n \sqsubseteq \overline{T}_{e_1}, T_{e_1} = \overline{T}_{e_1}, T_e \preceq \overline{T}_e\}$ $\cup \mathcal{C}(e_1)$
`uop` op e_1	$: \{T_{e_1} \bowtie T_e, T_e \bowtie T_{e_1}, T_e \preceq \overline{T}_e\} \cup \mathcal{C}(e_1)$
`addr &` e_1	$: \{T_e = D, T_{e_1} = D, T_e \preceq \overline{T}_e\} \cup \mathcal{C}(e_1)$
`bop` op e_1 e_2	$: \{T_{e_1} \bowtie T_e, T_e \bowtie T_{e_1}, T_{e_2} \bowtie T_e, T_e \bowtie T_{e_2}, T_e \preceq \overline{T}_e\}$ $\cup\, \mathcal{C}(e_1) \cup \mathcal{C}(e_2)$
`ecall` f`(`e_1`, ..., `e_n`)`	$: \{T_{e_i} \bowtie \overline{T}_e, T_e \bowtie \overline{T}_{e_i}, T_e \preceq \overline{T}_e\} \cup \bigcup_i \mathcal{C}(e_i)$

endcase

Figure 24: Constraints for expressions

A constant expression is always static, and a variable expression is described by the corresponding type variable.

Consider an index expression $e = e_1$`[`e_2`]`. The binding time type of e is determined by the binding time of the lefthand side expression e_1. Due to well-typedness, it must either be array type $[T]$ or D. Note, by the definition of the order $\sqsubseteq$, both possibilities are legal. The dependency constraints expresses that if either the lefthand side expression or the index expression is dynamic, then the whole expression e must be dynamic. This is in correspondence with the typing rules given in Figure 14. Since an array pointer cannot be lifted, the normal and lifted type of left-expression is set equal.

The rule for structure expression is similar. Note that if $\overline{T}_{e_1} = D$, then, by the definition of the partial order $\sqsubseteq$, T_e must equal D, since $\sqsubseteq$ is only satisfied if the componemts on the lefthand side are D.

The cases for application of unary and binary operators and external function calls are similar. The dependencies implies that if one of the arguments are non-static, then all arguments must be dynamic, and the whole expression suspended, *i.e.* the operation cannot be performed at specialization time.

The static binding time of operators and external functions is not needed. The (ordinary) type correctness of the program will assure that operators are only applied to expressions with "correct" binding time types.

Observe that the number of fields in a structure expression is needed. This is, however, part of the ordinary type information and can hence be found statically before the binding time analysis.

NOTATIONAL CONVENTION 5.1 *Given a solution* Θ *to a constraint set, we call a type assignment* τ *satisfying*

$$\forall v \in Id : \Theta(v) = T \Rightarrow \tau(v) = T$$

a constraint set environment for Θ.

$\mathcal{C}_{stmt}(s)$ = *case* s *of*

l : `assign` *lval* `=` *exp*	$: \{T_{lval} \bowtie\!\!\bowtie \overline{T}_{exp}, T_{exp} \bowtie\!\!\bowtie \overline{T}_{lval}, T_{lval} = \overline{T}_{lval}\}$ $\cup\, \mathcal{C}_{lexp}(lexp) \cup \mathcal{C}_{exp}(exp)$
l : `goto` m	$: \{\}$
l : `if (`*exp*`)` m n	$: \mathcal{C}_{exp}(exp)$
l : `return` *exp*	$: \{T_{\pi(f)} \preceq T_{exp}, T_{exp} \bowtie\!\!\bowtie T_{\pi(f)}\} \cup \mathcal{C}_{exp}(exp)$
l : `call` x `=` f'`(`e_1`, ..., `e_n`)`	$: \{T_{\pi(f')} \preceq \overline{T}_x, T_{e_1} \preceq T_{f'_1}, \ldots, T_{e_n} \preceq T_{f'_n}, T_x = \overline{T}_x,$ $T_{f'_1} \bowtie\!\!\bowtie T_{e_1}, \ldots, T_{f'_n} \bowtie\!\!\bowtie T_{e_n},$ $T_{e_1} \bowtie\!\!\bowtie T_{f'_1}, \ldots, T_{e_n} \bowtie\!\!\bowtie T_{f'_n}\} \bigcup_i \mathcal{C}_{exp}(e_i)$
l : `rcall` x `=` f'`(`e_1`, ..., `e_n`)`	$: \{T_{\pi(f')} \preceq \overline{T}_x, T_{e_1} \preceq T_{f'_1}, \ldots, T_{e_n} \preceq T_{f'_n}, T_x = \overline{T}_x$ $T_{f'_1} \bowtie\!\!\bowtie T_{e_1}, \ldots, T_{f'_n} \bowtie\!\!\bowtie T_{e_n},$ $T_{e_1} \bowtie\!\!\bowtie T_{f'_1}, \ldots, T_{e_n} \bowtie\!\!\bowtie T_{f'_n}\} \cup \bigcup_i \mathcal{C}_{exp}(e_i)$

endcase

where T_{f_i} denotes the type variable describing the i'th parameter of f.

Figure 25: Constraints for statements

PROPOSITION 5.1 *Let e be a Core C expression, and Θ a solution to $\mathcal{C}_{exp}(e)$. Assume e' is a completion of e annotated according to Θ, and τ is constraint set environment for Θ. Then it holds for some T:*

$$\tau, \gamma \vdash^{\text{EXP}} e' : T$$

i.e. e' is well-annotated.

Proof By case analysis of the inference rules. □

5.2.3 Constraint Set for Statements and Functions

Statements and functions generate some extra constraints corresponding to the type inference rules for these. The most restrictive is the assignment: the lefthand and righthand side expressions are checked for being non-static using the strong dependency constraint $\bowtie\!\!\bowtie$.

DEFINITION 5.7 (CONSTRAINTS FOR STATEMENTS)
Define for a statement s in function f the constraint set $\mathcal{C}(s)$ as shown in Figure 25. □

In the case of assignments, the last constraint expresses that a lefthand side expression cannot be lifted. This is assured by requiring the normal and the lifted binding time types to equal. The two constraints make both sides dynamic if one of the sides is non-static, cf. Example 4.4.

In the case of a return statement, the expression must be at least as dynamic as the return type of the function. Due to the definition of $\preceq$, no (partially) dynamic arrays or structures will be returned. Since the return type of a function is described by a

single type variable, all return statements are effectively made residual if just one return expression (in the function) is non-static.

The rules for `call` and `rcall` are equal. The actual expressions are related to the formal parameters. If an argument is non-static, the corresponding formal parameter must be D.

We have the following proposition.

PROPOSITION 5.2 *Let s be a Core C statement and $\mathcal{C}_{stmt}(s)$ the constraint set for s. Let τ, π be type assignments for solution Θ to the constraint set. Then for some type U:*

$$\tau, \pi, \gamma \vdash^{\text{STMT}} s : U$$

That is, the statement is well-annotated.

DEFINITION 5.8 (CONSTRAINTS FOR FUNCTIONS)
Let f be a function. The constraint set $\mathcal{C}_{fun}(f)$ is defined by

$$\begin{aligned}\mathcal{C}_{fun}(f) \;=\; & \bigcup_{s \in \mathcal{S}_f} \mathcal{C}_{stmt}(s) \\ & \cup \{T_{f_1} \bowtie\!\!\!\rhd T_{\pi(f)}, \ldots, T_{f_n} \bowtie\!\!\!\rhd T_{\pi(f)}\} \\ & \cup \{T_f \bowtie\!\!\!\rhd T_s \mid f \in \mathcal{RF}(f), s = \texttt{assign}\ \mathit{lval}\ \texttt{=}\ \mathit{exp} \in \mathcal{S}_f \wedge v \in \mathcal{GF}(f)\}\end{aligned}$$

where T_{f_i} is the type variable for the i'th formal parameter in f. □

These constraints expresses that if one of the parameters are dynamic, then the function is residual and must hence return a dynamic value, cf. the well-annotatedness rules for functions. The last set of constraints implies that if the function is residual and recursive, then all statements which assigns to non-local variables must be dynamic. By the rule for assignment, this also forces the assigned variables to be dynamic.

PROPOSITION 5.3 *Let p be a Core C program and $\mathcal{C}$ the constraints corresponding to p. Let Θ be a solution to $\mathcal{C}$ and τ, π be type assignments for to Θ. Then p is well-annotated according to Definition 4.12.*

Proof By case checking of the constraints generated by the functions. □

5.2.4 Normal Form Constraint Sets

We define *normal form* of constraint sets and give a set of *transformational* rules for finding it. Furthermore we characterize a normal form constraint set. Its form implies that a minimal solution can be found using a unification algorithm.

DEFINITION 5.9 (NORMAL FORM CONSTRAINT SETS)
Let $\mathcal{C}$ be a constraint set. $\mathcal{C}$ is in normal form if it consist solely of constraints of the form:

1. $T_1 \preceq T$
2. $[T_1] \sqsubseteq T$
3. $T_1 \otimes \cdots \otimes T_n \sqsubseteq T$
4. $T \rhd T_2$
5. $T \bowtie\!\!\rhd T_2$

where T_1 is static (S) or a type-variable, T_2 is dynamic (D) or a type variable and T is a type variable. A normal form constraint system contain no circles, that is, a term graph representation of the constraint set is non-circular. □

The next theorem states the existence of a normal form for all constraint sets and devise a constructive way of finding it.

THEOREM 5.1 (NORMAL FORM OF CONSTRAINT SETS) *Let $\mathcal{C}$ be a constraint set. $\mathcal{C}$ has a unique normal form $\mathcal{C}'$, and it can be found using the transformation rules in Figure 26 exhaustively.* □

Proof Observe that the constraints introduced by the transformation rules either will be eliminated by one of the other transformation rules, or are constraints on normal form. Hence, the transformation terminates. When none of the transformation rules are applicable, all constrains are normal form according to Theorem 5.1. □

It is easy to see that applications of the transformation rules preserve the set of solutions. That is, let the constraint system be $\mathcal{C}$ and let the normalized constraint system $\mathcal{C}'$ be obtained via the substitution Θ. Then

$$\mathrm{Sol}(\mathcal{C}) = \Theta \circ \mathrm{Sol}(\mathcal{C}').$$

The proof is by inspection of the transformation rules.

A main theorem of this chapter is the existence of a minimal solution to normal form constraint systems. The proof is constructive and can be used in an implementation. A solution Θ is said to be *ground* if no type variables remains after it is applied.

THEOREM 5.2 (MINIMAL SOLUTION TO NORMAL FORM CONSTRAINT SET) *Let $\mathcal{C}$ be a normal form constraint system. The system $\mathcal{C}$ has a unique minimal solution.* □

Proof Consider the constraint system as a set of equalities. By the unification Theorem [Robinson 1965] there exist a most general unifier U which solves $\mathcal{C}$. Let Θ be a substitution which maps all free variables in $U(\mathcal{C})$ to Θ. We claim that $U \circ \Theta$ is a minimal solution.

Neither the unifier U nor the substitution Θ maps any type variables to D so all dependencies are trivially satisfied, including the strong ones. Furthermore, it solves all lift and split inequalities by equality.

Assume Θ' is another solution to $\mathcal{C}$. Θ' either map a type variable to D or solve an inequality non-strictly, and is hence not less that the solution $U \circ \Theta$. Therefore, $U \circ \Theta$ is a minimal solution. □

The minimal solution of an arbitrary constraint system $\mathcal{C}$ is defined as follows. Let Θ be the substitution such that if $\mathcal{C} \Rightarrow \mathcal{C}'$ using the normalization rules in Figure 26 then $\Theta(\mathcal{C}) = \mathcal{C}'$. Let the solution to $\mathcal{C}'$ be Θ'. Then the solution to $\mathcal{C}$ is $\Theta \circ \Theta'$.

5.3 Binding Time Analysis Algorithm

Using the results from the previous sections, we are now in a position to describe the binding time analysis. It consists of four parts.

Definition 5.10 (Binding Time Analysis for Core C)
Let p be a Core C program and τ_0 an initial division. A division for p can then be computed as follows.

1. *Construct the constraint set $\mathcal{C}$ for p using Definition 5.8. Add constraints corresponding to the initial division τ_0.*
2. *Normalize the constraint system $\mathcal{C}$ to $\mathcal{C}'$ using the transformation rules given in Figure 26.*
3. *Find a minimal solution Θ to $\mathcal{C}'$ as outlined in the proof for Theorem 5.2.*
4. *Annotate the program p according to the solution Θ.*

□

The construction of a constraint set can be done in a single sweep through the program. In the constraint set normalization algorithm, some additional data structures are used. These should of course be initialized during the collection of constraints. In a concrete implementation, the syntax nodes in the abstract syntax tree for expression are most naturally used as "type-variables". As a side-effect, all expressions node are "automatically" decorated with the corresponding binding time type computed by the analysis.

Step two, normalization of the constraint set, can be done in almost linear time using Henglein's algorithm. It must, however, be extended slightly to treat constraints of form $T_1 \preceq T_2$ (shown in the paper) and strong dependencies $T_1 \bowtie T_2$. The latter can be done by a simple occur check in a (partial instantiated) type in the rules for equality constraints. The overhead will be dominated by the other operations.

Finding a minimal solution can be done using Robinson's unification algorithm as described in the existence proof for a minimal solution to a constraint set. Given an annotation of the expressions, the whole program can be annotated in a single sweep through the syntax.

Inequality rules

$C \cup \{S \preceq D\} \Rightarrow C$

$C \cup \{D \preceq T\} \Rightarrow C \cup \{T = D\}$

$C \cup \{T \preceq D\} \Rightarrow C \cup \{S \preceq T\}$

$C \cup \{[T_1] \sqsubseteq T, T_2 \preceq T\} \Rightarrow C \cup \{[T_1] \sqsubseteq T, T_2 = T$

$C \cup \{S \preceq T, T' \preceq T\} \Rightarrow C \cup \{S \preceq T, T' \rhd T\}$

$C \cup \{[D] \sqsubseteq D\} \Rightarrow C$

$C \cup \{[T] \sqsubseteq D\} \Rightarrow C \cup \{T = D\}$

$C \cup \{[T_1] \sqsubseteq T, [T_2] \sqsubseteq T\} \Rightarrow C \cup \{[T_1] \sqsubseteq T, T_1 = T_2\}$

$C \cup \{[T_1] \sqsubseteq T_2, T_2 \preceq T_3\} \Rightarrow C \cup \{[T_1] \sqsubseteq T_2, T_2 = T_3\}$

$C \cup \{D \otimes \cdots \otimes D \sqsubseteq D\} \Rightarrow C$

$C \cup \{T_1 \otimes \cdots \otimes T_n \sqsubseteq D\} \Rightarrow C \cup \{T_1 = D, \ldots, T_n = D\}$

$C \cup \{T_1 \otimes \cdots \otimes T_n \sqsubseteq T, T'_1 \otimes \cdots \otimes T'_n \sqsubseteq T\} \Rightarrow C \cup \{T_1 = T'_1, \ldots, T_n = T'_n, T_1 \otimes \cdots \otimes T_n \sqsubseteq T\}$

$C \cup \{T_1 \otimes \cdots \otimes T_n \sqsubseteq T, T \preceq T'\} \Rightarrow C \cup \{T_1 \otimes \cdots \otimes T_n \sqsubseteq T, T = T'\}$

$C \cup \{T_1 \otimes \cdots \otimes T_n \sqsubseteq T, T' \preceq T\} \Rightarrow C \cup \{T_1 \otimes \cdots \otimes T_n, T' = T\}$

Equality rules

$C \cup \{D = T\} \Rightarrow C \cup \{T = D\}$

$C \cup \{T = T\} \Rightarrow C$

$C \cup \{T = T'\} \Rightarrow S(C), S = \{T \mapsto T'\}$ where T type variable

Dependency rules

$C \cup \{T \rhd D\} \Rightarrow C$

$C \cup \{D \rhd T\} \Rightarrow C \cup \{T = D\}$

$C \cup \{T \bowtie D\} \Rightarrow C$

$C \cup \{D \bowtie T'\} \Rightarrow C \cup \{T' = D\}$

$C \cup \{[T_1] \bowtie T_2\} \Rightarrow C \cup \{T_1 \bowtie T_2\}$

$C \cup \{T_1 \otimes \cdots \otimes T_n \bowtie T\} \Rightarrow C \cup \{T_1 \bowtie T, \ldots, T_n \bowtie T\}$

Occur check

$\mathcal{C} \Rightarrow \mathcal{C} \cup \{T = D\}$ if $\mathcal{C}$ cyclic and T is on the cycle

Figure 26: Transformation rules for normalization of constraint systems

OBSERVATION 5.2 *A minimal completion of a Core C program will always annotate all* `goto` *as static. However, it may be desirable to annotate some* `goto` *statement dynamic for obtaining better sharing of code. We will apply the following heuristic. If the target of a* `goto` *is a dynamic* `if`*, we annotate the* `goto` *dynamic. This will typical prevent a loop to be "unfolded" once before the test is specialized. Practical experiments supports the heuristic.* END OF OBSERVATION

5.4 Example

For illustration we in this section give a complete example of binding time analysis of a program. As example program we use the (now well-known) *power* program.

```
int power(int n int x)
{
    int pow

    1: assign pow = 1
    2: if (n) 3 6
    3:    assign pow = pow * x
    4:    assign n = n - 1
    5: goto 2
    6: return pow
}
```

We use the following notation. Type variable T^x denotes the unique type variable associated with variable x. The type variable assigned to an expression will be subscripted with the expression. The notation should be clear from the context.

From the expressions, the following constraints are generated, cf. Definition 5.6.

1: $\{S = T_1, T_1 \preceq \overline{T}_1\}$
2: $\{T^n = T_n, T_n \preceq \overline{T}_n\}$
3: $\{T^x = T_x, T_x \preceq \overline{T}_x, T^{pow} = T_{pow}, T_{pow} \preceq \overline{T}_{pow},$
$T_{pow} \bowtie T_{pow*x}, T_{pow*x} \bowtie \overline{T}_{pow}, T_x \bowtie T_{pow*x}, T_{pow*x} \bowtie \overline{T}_x, T_{pow*x} \preceq \overline{T}_{pow*x}\}$
4: $\{T^n = T_n, T_n \preceq \overline{T}_n, S = T_1, T_1 \preceq \overline{T}_1\}$
5: $\{\}$
6: $\{\}$

The statements adds the following constraints, cf. Definition 5.7.

1: $\{T_{pow} \bowtie \overline{T}_1, T_1 \bowtie T_{pow}\}$
2: $\{S \preceq \overline{T}_n\}$
3: $\{T_{pow} \bowtie \overline{T}_{pow*x}, T_{pow*x} \bowtie T_{pow}\}$
4: $\{T_n \bowtie \overline{T}_{n-1}, T_{n-1} \bowtie T_{pow}\}$
5: $\{\}$
6: $\{T_{power} \preceq \overline{T}_{pow}, T_{pow} \bowtie T_{power}\}$

Finally, the *power* function adds the constraints, cf. Definition 5.8:

$$\{T_n \rhd\!\!\rhd T_{power}, T_x \rhd\!\!\rhd T_{power}\}$$

The initial division is that n is static but x dynamic. This is expressed in the constraints:

$$\{S = T_n, D = T_x\}$$

which ends the generation of the constraint set.

Applying the normalization rules in Figure 26 yields the following normal form constraint set:

$$\{S \preceq \overline{T}_1, S \preceq \overline{T}_n, S \preceq \overline{T}_n,$$
$$S \rhd\!\!\rhd \overline{T}_{n-1}, S \rhd\!\!\rhd \overline{T}_{n-1}, T_{n-1} \rhd\!\!\rhd \overline{T}_1, T_{n-1} \rhd\!\!\rhd T_n, T_n \rhd\!\!\rhd \overline{T}_{n-1}, T_{n-1} \rhd\!\!\rhd T_n\}$$

and substitution

$$S = \{T_1 \mapsto S, \overline{T}_1 \mapsto D, T_n \mapsto S, T_x \mapsto D, T_{pow} \mapsto D, \overline{T}_{pow} \mapsto D,$$
$$T_{pow*x} \mapsto D, \overline{T}_{pow*x} \mapsto D\}$$

where the $\overline{T}_1$ variable is the one from line 1.

It is easy to see that the minimal solution is the substitution mapping $\overline{T}_1$, T_{n-1}, $\overline{T}_{n-1}$, T_n and $\overline{T}_n$ to S.

From this we derive the following (well-annotated!) Core C program:

```
int power(int n int x)
{

    int pow

    1: assign pow = lift(1)
    2: if (n) 3 5
    3:    assign pow = pow * x
    4:    assign n = n - 1
    5: goto 2
    6: return pow
}
```

which clealy is a well-annotation and is minimal.

5.5 Terminating Divisions

Consider the following two-level Core C program.

```
int foo()
{
    int n

    1: assign n = 0
    2: if (n > ⟨dynamic expression⟩) 3 4
    3: return lift(n)
```

```
    4: assign n = n + 1
    5: goto 2
}
```

It is easy to see that the program is well-annotated according to Definition 4.12. Unfortunately, if the program is specialized, the specializer will not terminate even though a normal execution of the program *will* terminate. Clearly, if n is made dynamic, the specialization process terminates.

In this section we will discuss the imperfect termination properties which our program specializer shares with almost all other program specializers.

Definition 5.11 (Terminating Division)
Let τ be a division. τ is said to be a terminating division if the specializer is guaranteed to terminate when applied to the program annotated according to τ. □

The problem of finding a terminating division has been addressed by Jones [Jones 1988] and more recently, in another framework, by Holst [Holst 1991].

The normal "conservative" way of ensuring termination of the program specialization, is to require at every specialization point that at least one variable must be strictly decreasing (in some ordering). This corresponds to "recursive" descent in parsing terminology. At least in the functional language world, this requirement has turned out to be too restrictive in many cases.

The more liberal termination analysis developed in [Jones 1988] and [Holst 1991] can briefly be described as follows.

> Collect all *transitions* in the program. The requirement is then: at every transition where a (static) variable increases there must also be a decresing (static) variable[2].

In our setting, things are considerably more complicated, though.

Assume for instance that a function is specialized to an array. In what sense is the array "decreasing"? At compile-time, the only possibility seems to treat the whole array as one entity. Furthermore, the presence of pointers and side-effect at specialization time inevitably demands an alias analysis. We have a strong feeling that a termination analysis must be overly conservative in order to work.

5.6 Summary

We have described an automatic binding time analysis for Core C founded on the solution of *constraint* sets. The analysis is capable of finding a division of a program's variables into specialization time and run time such that the program can be well-annotated according to the well-annotatedness rules defined in Chapter 4. The analysis can be implemented to run in almost linear time.

In the next chapter the methods of the binding time analysis is applied to *untagging analysis*. In Chapter 8, Section 8.2 we discuss other applications.

[2] we have tacitly omitted details about *self-dependency etc*

Chapter 6

Untagging Analysis

When a self-interpreter in a typed language is specialized to a program, the residual program inherit its types from the self-interpreter. For example, let *SelfInt* be the Core C self-interpreter and suppose it is specialized to a program p:

$$[\![spec]\!]_{\mathcal{C}}(\overline{SelfInt}^{[Fun]}, \overline{\overline{p}^{[Fun]}}^{Val}) \Rightarrow \overline{p'}^{[Fun]}.$$

The variables in p' will be of type `Value`, since the self-interpreter uses `Value` for representation of values. Hence, the types in p' are too *general*, and further, *mix* is not as optimal as desired.

In this chapter we develop an analysis for *simplification* of the types in residual programs, and *removal* of unnecessary tag/untagging operators. The precise purpose of the analysis is stated in Section 6.1.

The problem of too general types in connection with partial evaluation of typed languages, and especially in connection with self-application, was apparently first addressed by Launchbury in his self-applicable partial evaluator for a subset of the typed language LML [Launchbury 1991]. The problem in Core C is of a similar nature, but we formulate it in another way.

The problem of untagging is closely related to *dynamic typing* [Henglein 1991a] and *partial type inference* [Thatte 1988], [Gomard 1990][1]. In *dynamic typing* the goal is to remove as many tag-operations in a dynamically typed program as possible on the basis of static typing information derivable. Our problem is similar. We have a program (which we known is perfectly "well-typed") but it inject and project values in and out of the `Value` type using tag-operations. How many of these can be safely removed?

The chapter is organized as follows. First we take a close look at the problem and state the purpose of the untagging analysis. Then, in Section 6.2, we give the analysis using a *constraint set* based inference analysis similar to the binding time analysis in Chapter 5. This demonstrates the generality of the constraint set based approach to program analyses.

[1] Thatte use the term "partial type" as a generalization of the strong typing discipline, whereas Gomard use the term in the meaning: "how much type information can statically be derived from an untyped program".

6.1 The Untagging Problem

Assume that we (once again!) specialize the self-interpreter to the `power` program. A residual program of a form resembling the program below will then be generated[2].

```
Value power(Value n Value x)
{
    Value pow

    1: pow = int_to_val(1);
    2: if (val_to_int(n)) 3 6
    3:    pow = int_to_val(val_to_int(pow) * val_to_int(x))
    4:    n = int_to_val(val_to_int(n) - 1)
    5: goto 2
    6: return pow
}
```

For illustrating purposes we have used the same names as in the original program.

Observe that all variables and functions are of type `Value`. This can be traced back to the use of `Value` arrays for representation of values in the self-interpreter. The functions `int_to_val()` *etc.* are injections and projections on `Value`. For instance, the minus operator takes integer arguments, and hence the integer part of `pow` and `x` must be projected out before − can be applied. The result must be injected back before store in `pow`.

What we *want* from the untagging analysis is to determine that the variables `n`, `x` and `pow` all are integer variables, and hence that all the injections and projections can be eliminated.

What we *know* is that the given program *is* well-typed. Hence, if we remove no projections and injections and change the type of no variables, then we have a (type) correct program. However, we do *not* know for sure that the given program is completely statically typed, that is, that no variable both represents integer and character values, say. A compiler, generated by self-application, *do* need the `Value` type for representation of specialization time values. In other words, we can not always expect the analysis to remove the `Value` type completely.

We have the following conjecture.

CONJECTURE 6.1 *If the program p uses no* `Value` *type, the residual program p' can be untagged to a program without the* `Value` *type, and all tagging and untagging operations can be removed.*

The conjecture is strongly dependent upon the optimality of *mix*. If *mix not* is optimal, removal of all `Value` variables cannot be expected.

Assume to types T_1 and T_2 such that T_1 is *coercible* to T_2. For example, T_1 might be `int` and T_2 `Value`. A coercion from a type to a more general type is called a *positive* coercion. The other way around is called a *negative* coercion. The positive coercion

[2]We have written the program in Core C for notational convenience

function between T_1 and T_2 is `int2val()` and the negative `val2int()`. A positive coercion correspond to a *tagging* operation, the negative operation is an *untagging* operation. A run-time type error (may) occur if `val2int()` is applied to a `Value` value tagged as a character.

The general idea is now to *ignore* all *mix*-produced coercions and to infer which coercions really are needed via type inference. The information is collected in a *constraint set*.

6.2 Constraint Based Untagging Analysis

The *untagging analysis* is formulated as a *constraint set* analysis similar to the binding time analysis from the previous chapter.

6.2.1 Untagging Constraint Sets

In the untagging analysis, the constraint set collects *type information*—the type of expressions.

Definition 6.1 (Untagging Constraint Set)
A constraint set is a (multi)set of constraints of form

$$T_1 = T_2 \qquad T_1 \bowtie T_2$$

where T_i is a type variable over Core C types. □

The symbol $\bowtie$ represent coercion. The symmetric symbol indicates that both positive and negative coercions are possible. In Core C, the coercions are between base types and pointers, and `Value`.

Definition 6.2 (Solution to Constraint Set)
Let $\mathcal{C}$ be a coercion set. A solution to $\mathcal{C}$ is a substitution $\Theta : TypeVar \rightarrow Type$ such that

1. *For all $T_1 = T_2 \in \mathcal{C}$: $\Theta(T_1) = \Theta(T_2)$*
2. *For all $T_1 \bowtie T_2 \in \mathcal{C}$: $\Theta(T_1) \bowtie \Theta(T_2)$*

and Θ is the identity on other type variables no occuring in $\mathcal{C}$ □

A normal form constraint set is defined as in the binding time analysis. In this analysis, the normal form consist of constraints of form $T_1 = T_2$ and $T_1 \bowtie T_2$. The *normal form* can be found by exhaustive applications of transformation rules. The transformations rules for the untagging analysis is shown in Figure 27.

Theorem 6.1 (Minimal Solution to Constraint System) *Let $\mathcal{C}$ be a constraint system. Then $\mathcal{C}$ has a normal form $\mathcal{C}'$, and the $\mathcal{C}'$ has a minimal solution.* □

Coercion rules	
$C \cup \{T \bowtie T\}$	$\Rightarrow C$
$C \cup \{T_1 \bowtie T_2\}$	$\Rightarrow C$ T_1, T_2 base types and a legal coersion
$C \cup \{T_1 \bowtie T, T_2 \bowtie T\}$	$\Rightarrow C \cup \{T = \texttt{Val}, T_1 \bowtie \texttt{Val}, T_2 \bowtie \texttt{Val}\}$
$C \cup \{[T_1] \bowtie [T_2]\}$	$\Rightarrow C \cup \{T_1 = T_2\}$
$C \cup \{T_1 \otimes \cdots \otimes T_n \bowtie T'_1 \otimes \cdots \otimes T'_n\}$	$\Rightarrow C \cup \{T_1 = T'_1, \ldots, T_n = T'_n\}$
Equality rules	
$C \cup \{T = T\}$	$\Rightarrow C$
$C \cup \{T_1 = T_2\}$	$\Rightarrow C \cup \{T_1 = T_2\}$ T_2 a type variable
$C \cup \{T_1 = T_2\}$	$\Rightarrow \Theta(C), \Theta = \{T_1 \mapsto T_2\}$ T_1 a type variable
Occur check	
C cyclic graph	$\Rightarrow C \cup \{T = \texttt{Val}\}$ T on cycle in C

Figure 27: Transformation rules for constraints

Proof Similar to the proof of the corresponding theorem in Chapter 5. □

6.2.2 Constraint Set Construction

To each expression e we assign two unique type variables. The first, T_e, expresses the type of e. The latter, $\overline{T}_e$, denotes the *coerced* type. Another way of thinking of the two type variables is that T_e is the *real* type of e while $\overline{T}_e$ is the *wanted* type of e. For instance, in an `if` statement, an expression may be of type `Value` but the wanted type is `int`. Hence, a negative coercion `val2int()` must be applied.

To each variable v we assign a type variable T_v as well to each function.

The construction of a constraint set for a Core C program is defined below. We assume the type operator $\mathcal{O}$ provide type information about primitive operators and external functions.

Definition 6.3 (Constraints for Expressions)
Let e be a Core C expression. Define the constraint set $C_{exp}(e)$ for e inductively as shown in Figure 28. □

The type of a constant is given by the (unspecified) function *type*. The type of a variable v is determined by the associated type variable T_v.

In the case of an index expression, the wanted type of the lefthand expression is a pointer type. The index expression must be an integer type. For structures, the subexpression must possess a structure type.

The applications of unary and binary operators and external functions are all the same. The type of the operator (function) gives information about the *wanted* type of the arguments, and the type of the value actually delivered. In the case of an address operator, the type of the subexpression must be a pointer type.

Statements adds the following constraints to the constraint set.

$\mathcal{C}_{exp}(e)$ = *case* e *of*

`cst` c	$: \{T_e = type(c), T_e \bowtie \overline{T}_e\}$
`var` v	$: \{T_e = T_v, T_e = \overline{T}_e\}$
`index` e_1`[`e_2`]`	$: \{[T_e] = \overline{T}_{e_1}, \mathsf{int} = \overline{T}_{e_2}, T_e \bowtie \overline{T}_e\} \cup \mathcal{C}_{exp}(e_1) \cup \mathcal{C}_{exp}(e_2)$
`struct` e_1`.`y	$: \{T_1 \otimes \cdots \otimes T_e \otimes \cdots \otimes T_n = \overline{T}_{e_1}, T_e \bowtie \overline{T}_e\} \cup \mathcal{C}_{exp}(e_1)$
`addr &` e	$: \{T_e = [\overline{T}_{e_1}], T_e \bowtie \overline{T}_e\} \cup \mathcal{C}_{exp}(e)$
`uop` op e_1	$: \{\overline{T}_{e_1} = T_1, T_e = T, T_e = \overline{T}_e\} \cup \mathcal{C}_{exp}(e_1)$ where $\mathcal{O}(op) = T_1 \to T$
`bop` op e_1 e_2	$: \{\overline{T}_{e_1} = T_1, \overline{T}_{e_2} = T_2, T_e = T, T_e \bowtie \overline{T}_e\} \cup \mathcal{C}_{exp}(e_1) \cup \mathcal{C}_{exp}(e_2)$ where $\mathcal{O}(op) = T_1 \times T_2 \to T$
`ecall` f `(`e_1`, ..., `e_n`)`	$: \{\overline{T}_{e_i} = T_i, T_e = T, T_e \bowtie \overline{T}_e\} \cup \bigcup_i \mathcal{C}_{exp}(e_i)$ where $\mathcal{O}(op) = T_1 \times \cdots \times T_n \to T$

endcase

where $\mathcal{O}$ is the operator type oracle.

Figure 28: Constraint for expressions

$\mathcal{C}_{stmt}(s)$ = *case* s *of*

l : `assign` $lexp$ `=` exp	$: \{T_{lexp} = \overline{T}_{exp}, T_{lexp} = \overline{T}_{lexp}\} \cup \mathcal{C}_{exp}(lexp) \cup \mathcal{C}_{exp}(exp)$
l : `goto` m	$: \{\}$
l : `if (`exp`)` m n	$: \{\mathsf{int} = \overline{T}_{exp}\} \cup \mathcal{C}_{exp}(exp)$
l : `return` exp	$: \{T_f = \overline{T}_{exp}\} \cup \mathcal{C}_{exp}(exp)$
l : `call` x `=` f `(`e_1`, ..., `e_n`)`	$: \{T_x = \overline{T}_x, T_x = T_f, T_{v_1} = \overline{T}_{e_1}, \ldots, T_{v_n} = \overline{T}_{e_n} \cup \bigcup_i \mathcal{C}_{exp}(e_i)$ where T_{v_i} type variable for i'th parameter in f

endcase

Figure 29: Constraints for statements

Definition 6.4 (Constraints for Statements)
Let s be a Core C statement. The set of constraints generated by s (in function f) is given as shown in Figure 29. □

In an assignment, the tagged type of the righthand side expression must equal the type of the lefthand side expression. Note that an tagging operator can not be applied on a lefthand side.

In the case if an `if` statement, the type of the test-expression must be `int`. In `return` statements, the tagged type of the expression must equal the return type of the function.

In function calls, the assigned variable cannot be tagged. The actual expression is matched against the formal parameter of the called function.

Definition 6.5 (Constraint Set for Core C Program)
Let p be a Core C program with functions $\mathcal{F}$. The set of constraints $\mathcal{C}(p)$ for p is then given by

$$\mathcal{C}(p) = \bigcup_{f \in \mathcal{F}} \bigcup_{s \in f} \mathcal{C}_{stmt}(s)$$

where $\mathcal{C}_{stmt}$ is defined above. □

6.3 Untagging Analysis Algorithm

A solution to a constraint set can be found using similar techniques as in the binding time analysis. The constraint set is first *normalized*, and then a minimal solution can be found using a unification algorithm. Given the type information on each expression, it is easy to determine where corecions must be applied. We leave the details unspecified.

6.4 Example

As an illustration of the untagging analysis, we give a complete example of untagging the *mix*-produced `power` program from first of this section.

```
Value power(Value n Value x)
{
    Value pow

    1: pow = int_to_val(1);
    2: if (val_to_int(n)) 3 6
    3:    pow = int_to_val(val_to_int(pow) * val_to_int(x))
    4:    n = int_to_val(val_to_int(n) - 1)
    5: goto 2
    6: return pow
}
```

We use the same notation as in the example of the binding time analysis. For expressions, the following constraints are generated.

1: $\{T_1 = \mathsf{int}, T_1 \bowtie \overline{T}_1\}$
2: $\{T_n = T^n\}$
3: $\{T_{pow} = T^{pow}, T_{pow} = \overline{T}_{pow}, T_x = T^x, T_x = \overline{T}_x, T_x \bowtie \overline{T}_x,$
$\overline{T}_{pow} = \mathsf{int}, T_{pow} \bowtie \overline{T}_{pow}, \overline{T}_x = \mathsf{int}, T_x \bowtie \overline{T}_x,$
$T_{pow*x} = \mathsf{int}, T_{pow*x} = \overline{T}_{pow*x}\}$
4: $\{T_n = T^n, T_n = \overline{T}_n, T_n \bowtie \overline{T}_n, T_1 = \mathsf{int}, T_1 \bowtie \overline{T}_1,$
$\overline{T}_n = \mathsf{int}, \overline{T}_1 = \mathsf{int}, T_{n-1} = \mathsf{int}, T_{n-1} = \mathsf{int}, T_{n-1} \bowtie \overline{T}_{n-1}\}$
5: $\{\}$
6: $\{T_{pow} = T^{pow}\}$

The following constrains are due to statements.

1: $\{T_{pow} = \overline{T}_1\}$
2: $\{\overline{T}_n = \mathsf{int}\}$
3: $\{T_{pow} = \overline{T}_{pow*x}\}$
4: $\{T_n = \overline{T}_{n-1}\}$
5: $\{\}$
6: $\{T_{power} = \overline{T}_{pow}\}$

Normalization by applying the transformations rules in Figure 27 yields the following constraint set.

$$\{T_{pow} \bowtie \mathsf{int}, T_x \bowtie \mathsf{int}, T_n \bowtie \mathsf{int}, T_n = \overline{T}_{n-1}\}$$

and substitution

$$S = \{T_1, \overline{T}_1, \overline{T}_n, \overline{T}_x, T_{pow}, \overline{T}_{pow}, T_{power}, T_{pow*x}, T_{n-1}, \overline{T}_{pow*x} \mapsto \mathsf{int}\}.$$

Choosing the minimal solution by solving all coercions by equality, *i.e.* mapping the free variables to `int`, give the wanted, completely untagged *power* program.
We have in this example imposed no restrictions on the input parameters `n` and `x`. This may be necessary in order to prevent interfacing problems with the calling functions.

6.5 Summary

In this chapter we have developed an *untagging analysis* based on constraint set solution. The use of constraints captures in a natural way the requirement for insertion of coercions, and due to the fast normalization and unification algorithms, the analysis is tractable.

Chapter 7

Experiments

We have made a proto type implementation of the C program specializer and applied it in a number of experiments. Several of the examples are stolen from Pagan's book [Pagan 1990] and hence shows that automation of his hand methods is possible.

Section 7.1 provides an overview over the implementation. Some of the algoprithm from Chapter 3 is reconsidered, and various tricks and hacks described.

In Section 7.2 some of the experiments we have done with C-Mix are described. Benchmarks and assessements are given. We specialize a general scanner, we demonstrate removal of a complete layer of self-interpretation, and compiler genereration by self-application.

Since the thesis version of this report was handed in, an almost completely new implementation of the system has been made. Especially the specializer has been rewritten, and this has improved self-application considerably.

7.1 C-Mix Implementation

Central parts of the C program specializer developed in theory in this thesis has been implemented in C—approximately 12000 lines of code. Currently it consist of a parser from almost full ANSI C to Core C, the specializer (including Core C parser, memory managent, expression evaluation and reduction, *etc*) and an unparser. The binding time analysis and untagging analysis has not been implemented[1], and hence the system is not fully automatic "as is". Furthermore, since livevariable analysis is important, this should also be considered as a part of an automatic system.

The unparse is pretty simple and is mostly a simple transformation from Core C to C. This means the residual programs are very much "flow-chart" like and open up for a lot of optimizations. However, we believe this is unimportant as the C compilers can do this transformations. A pretty unparse would be nice when inspecting the residual programs, though!

We have done some experiments with an implementation of binding time constraint solving in a Miranda[2] implementation, and is convinced that the binding time analysis is

[1] At time of writing, implementation of the binding time analysis is in progress

[2] Miranda is a trademark of Research software ltd.

capable of *e.g.* annotating the self-interpreter sufficiently statically for specialization.

Implementation of the untagging analysis should be easy given the binding time analysis.

7.1.1 Program Representation

In Chapter 2 we outlined a program representation in the form "an array (of functions) of an array (of statements)". This is clearly insufficient: what about expressions? The problem is that we seemingly have no pointers at out disposal leaving us with a representation with arrays and structures. Inconvenient and inefficient. We have exploited another trick.

A new base type `Expr` is introduced into the treated subset of C. The specializer treat the `Expr`-type just as `int` and `double`, that is, it knows nothing about the *internal* of such a type. In reality, `Expr` is a structure for representation of expressions. Since the specializer never inspects the internal of an `Expr`, it is possible to link `Expr` structures by pointers. Hence, suitable and desirable syntax tree for expressions have been introduced through the "backdoor"!

If `Expr cmix_make_int(int)` is a function taking an integer constant, then

```
Expr e = cmix_make_int(1)
```

will assign the expression "1" to e.

7.1.2 Lifting Basetypes

The solution of the representation problem introduces another problem: lifting of basetypes. Recall that the lift operator `@` means: evaluate the expression and build a residual constant. This present no problems in the case of integer and string expressions, say, but lift may also get applied to an `Expr`. This may for example happen during self-application to an interpreter, as parts of the interpreter is needed in the residual program (for example variable declarations, which conveniently also is represented in `Expr` expressions).

For lifting `Expr`essions we have introduced a library of syntax constructor functions such as `cmix_make_int()`. Hence, an expression `x[2]` may be lifted to

```
cmix_make_index(cmix_make_id("x"), cmix_make_int(2)).
```

These functions should be easy to spot in the listing in the appendices ....

7.1.3 The Specializer

The specializer, `spec` (472 lines), is more or less a straighforward realization of the algorithms in Chapter 3. Several important improvements are incorporated, however.

Consider the pending list. At self-application time it will be dynamic since the values of the program (to which the specializer is specialized to) is unknown. Similar for the code list; it will also be dynamic. This is not surprising: in a compiler the code list is definitely needed!

It is therefore possible to use an *external* representation of the pending and code lists, accessible via external function calls. For example, an element is inserted into the "current" pending by the means of the call

`insert_pend(`⟨*prog.point*⟩`, pstore, lstore, gstore);`

where `pstore`, `lstore`, `gstore` is arrays for parameters, local and global variables, respectively. The call

`make_assign(`⟨*l-exp*⟩`,` ⟨*exp*⟩`)`

generate an assignment and adds it to the accumulated residual statements.

This "trick" slows down the specializer as many function calls are introduced, but it improves self-application considerably as the representation problem of the pending array is avoided.

Consider again the pending loop as outlined in Chapter 3, here formulated with an external representation of `pending`.

```
while (!pending_empty())
   {
       /* Restore computation state according to pending */
       pp = pending_pp();
       pstore = pending_param();
       lstore = pending_local();
       gstore = pending_global();
       rpp = pending_processed();
       /* Perform two-level execution */
       while (pp)
          ...
   }
```

The function `pending_processed()` takes out the first element in `pending` and marks it as "processed".

Recall from above that at self-application time, `pending` is dynamic. This implies that `pp` also is dynamic—a highly undiesirable consequence. This means that almost nothing can be done at specialization time, especially, the specialization pf program points in the program to which the specializer is specialized is hindered.

The trick is to observe that `pp` only can assume *finite* many *static* values: the program points in the input program. This can be exploided by performing a program point lookup under static control.

```
while (!pending_empty())
   {
       /* Restore computation state according to pending */
       pp0 = pending_pp();
       pstore = pending_param();
       lstore = pending_local();
```

```
        gstore = pending_global();
        rpp = pending_processed();
        /* Lookup program point pp0 under static control */
        for (pp = 1; pp != pp0; pp += 1);
        /* Perform two-level execution */
        while (pp)
           ...
    }
```

Notice that the variable `pp` now is static, and hence the two-level execution can be performed at specialization time.

It was the same property that was used in the specialization of static side-effects under dynamic control. In fact, in partial evaluation it is often known from knowledge of the subject program that a variable only can assume finite many static values, but the particular one is dynamically determined. The outcome of specialization to all the static values is residual code of the following form:

```
switch (pp)
   {
    case One: ...
    case Two: ...
    ⋮
   }
```

Specialization to finite many values is applied so often in partial evaluation that it has been named "*the* trick".

There are still room for improvements of the program point specialization, however. Applying the above trick will result in a separate specialization of every program point in the input program—this is clearly too much. The key observation is that only *target* points need to be specialized, that is, targets for `goto`, `if`, the statement after dynamic, non-recursive `call`, and the first program point in a function. A bit of reprogramming can reduce the amount of specialization.

```
/* Lookup program point pp0 under static control */
for (pp = 1; pp != pp0; pp += 1)
   while (!spec_point(pp)) pp += 1;
/* Perform two-level execution */
```

The function `spec_point()` returns 1 if the program point is a specialization point.

Finally a note about transition compression. Recall that static transitions are compressed at specialization time. This is usually a good strategy, but some sharing is lost. Consider the overall structure of the specializer (here in pseudo Core C).

```
   /* Initialize */
   ...
   /* Pending loop */
L: if (!pending_empty())
     ...
   goto L;
```

The result of the (implicit) static transition from the initialization part to the `if` of the pending loop is, that the test expression will come out twice in the residual program. By insertion of a dynamic `goto L` as the last part of the initialization, this is prevented. This trick has been introduced several places in the specializer whereby better self-application is gained.

7.1.4 The Memory Management

As discussed in Chapter 3, copying and restoring of the specialization time stores. For this, a *tagged* memory representation is needed, similar as in copying collectors. The store is an array of `Value` structures. Each structure is equipped with a tag field, but for comparison of specialization time stores, the length of arrays is also needed. As it in general is undesirable to tag pointers with the length of the array they point to, we use the location -1 to store the length information. This is one of the reasons to our ban of static pointers to the middle of arrays—the length is needed!

7.2 Experiments with C-Mix

The C-Mix system has been applied in a number of experiments. In this section we describe some of these, provide benchmarks and assessments, and discuss its performance. All emprical experiments have been performed on a Sun SparcStation II. Reported time is user seconds (measured by the UNIX `time(1)` command), and code size is number of lines. The lenght of user written programs include comments; residual programs have been formatted by the `indent` UNIX program.

The test of the specializer can be found as Appendix B, and an annotated Core C version as Appendix C. Below we use `cogen` for the specialized version of `spec` with respect to `spec`.

Program run	Run time	Code size
$[\![spec]\!](spec, spec)$	4.8	2049

As apparent above, we will in this chapter be a bit sloppy and omit program and value representations. These can be recovered from Chapter 1.

7.2.1 Specialization of a General Scanner

We have implemented the general scanner by Pagan [Pagan 1990, 4.3] in C, and specialized. The scanner take as input a token description and stream of characters. By specialization to the token table, a lexical analysis tailored to the given set of tokens is produced. For a description of the algorithm and the program we refer to [Pagan 1990]. For direct comparison with the results Pagan has obtained, we use the same input data. The drive loop is given in Figure 30.

The token specification defines the following eight tokens

:, :=, :=:, :/=:, =, /, /=, %%

```
int scan(void)
{
    int p,   /* state */,
        tok, /* last token recogniced */
        ch;  /* next character */

    p = 1;
    while (p)
      if (trietable[p].ch == ch)
        {
            tok = trietable[p].tok;
            ch = getchar();
            p = trietable[p].next;
        }
      else
        p = trietable[p].alt;
    return tok;
}
```

Figure 30: The driver loop in the general scanner

the scanner table can be found in the book. Input string is "::/=::=::=//=%%" repeated 64000 times (Pagan use this pattern 4000 times, but our computers are a bit faster!).

Specialization yields a lexical analysis given in Figure 31. The program (fragment) is a generated by the system[3] except that we have removed some superfluous labels. This is what a handwritten scanner would look like!

By specialization of the specializer to the scanner, a lexical analysis generator (`lex`) is generated. Input is a token specification and output a lexical analysis.

The following benchmarks have been found.

Program run	Run time		Code size	
	time	ratio	size	ratio
$[\![scan]\!](Table, Input)$	16.1		65	
$[\![scan_{Table}]\!](Input)$	13.4	1.2	186	0.3
$[\![spec]\!](scan, Table)$	0.1		474	
$[\![spec_{scan}]\!](Table)$	0.1	1.0	344	1.4
$[\![spec]\!](spec, scan)$	0.9		474	
$[\![cogen]\!](scan)$	0.3	3.0	2049	0.2

The seemingly little speedup achieved by specialization of the general scanner to a particular token description is mainly due to the size otf the table. If a larger table involving more backtracking operations, larger speedup is obtained. The code blowup is

[3]It has been formatted by the indent program, though

```
int
scan_2_1 (void)
{
   if (ch == '!')
     {
        cmix_endconf = 1;
        return 0;
     }
   else
     {
        if (':' == ch)
          {
             ch = getchar ();
             if ('=' == ch)
               {
                  ch = getchar ();
                  if (':' == ch)
                    {
                       ch = getchar ();
                       cmix_endconf = 2;
                       return 3;
                    }
                  else
                    {
                       cmix_endconf = 3;
                       return 2;
                    }
               }
             else
               {
                  if ('/' == ch)
                    {

(186 lines in total)
```

Figure 31: Automatically generated lexical analysis

not surprisingly, but it is absolutely tolerable. The code blowup depend, however, on the size of the table. Thus, larger speedup, larger programs!

Notice that no performance improvement is gained by running the compiler ("lex") compared to direct specialization of the general scanner to a table. This is probadly due to the size of the scanner program; only 65 lines of code. The overhead in the compiler dominates the run times.

Pagan obtain a speedup of 1.7 by specialization of the scanner to the token table. This somewhat larger speedup is probadly due to his use of Pasca—we strongly suspect his PC Pascal compiler does not perform the optimizations the UNIX C compiler does.

7.2.2 Generating a Power Compiler

A running example in this thesis has been the `power` program. We will now examine the structure of a `power` compiler, that is, a program which, given n, deliver a program computing x to the nth. The compiler can be generated by self-application of the specializer with respect to `power`, or by applying `cogen`. The result is shown in Figure 32. We have (by hand) "sugared" the code: a `while` loop has replaced a `if-goto`, and superfluous labels have been removed. The power compiler as generated by C-Mix can be seen in Appendix D.

Notice the very natural structure of the compiler. After the initialization of the data structures (that is, the pending list and the code list), a compile time loop generates assignments `pow = pow * x` n times. The last statement to be generated is a `return pow`. A handwritten compiler would have the same structure, and delighted we cite Romanemko "A Compiler Generator Produced by a Self-Applicable Specializer Can Have a Surprisingly Natural and Understandable Structure" [Romanenko 1988]!

The only blemish of the compiler is the many tagging functions (`val_to_int()` and `int_to_val()`. The untagging analysis from Chapter 6 would be helpful here, and a lowering of the types would speedup the compiler.

7.2.3 Programming Language Implementation

The following example is stole from [Pagan 1988] where a generating extension is derived by hand. We show that partial evaluation can do it automatically.

An interpreter for a "poolish-form" assembler language has been implemented. The interpreter simulates a primitive stack machine with instructions like `STORE`, `LOAD`, and `ADD`. The set of instructions can be seen in the interpreter which can be found as Appendix E. The interpreter take as argument an array of machine instructions op-codes. The vairable `s` is the stack and the array `stg` is storage for program variables.

Consider the interpreter. The use of the stack is static, and hence it can be split. On the other hand, the storage `stg` cannot be split, which one might suspect, since the language has dynamic variable reference.

The "poolish-form" program given to the interpreter reads a number n and computers the first n primes. The algorithm used is shown below.

```
read n;
pc := 2; write 2; write 3;
loop
    exitif pc >= n;
    loop
        p := p + 2;
        nd := 0; odd := 1;
        loop
            odd := odd + 2;
            exitif odd >= p;
            if p / odd * odd = p then
                nd := nd + 1;
```

```
            fi
        end;
        exitif nd <= 0
    end;
    write p;
    pc := pc + 1
end
```

Specialization of the interpreter to the program yields a primes program in C. This can be seen in Appendix F. Notice it possess an almost flow-chart structure, and can be optimized using standard flow-analysis techniques. The result will be a "machine-code" program.

Below benchmarks are measured.

Program run	Run time		Code size	
	time	ratio	size	ratio
$[\![Int]\!](Primes, 500)$	61.9		123	
$[\![Int_{Primes}]\!](500)$	8.9	7.0	118	1.0
$[\![spec]\!](Int, Primes)$	0.6		474	
$[\![spec_{Int}]\!](Primes)$	0.5	1.2	760	0.6
$[\![spec]\!](spec, Int)$	2.2		474	
$[\![cogen]\!](Int)$	0.7	3.1	2049	0.2

The figures shows that compilation of the primes program to C pay off—a speedup of 7 is obtained. As the previous example, using the stand-alone compiler compared to the specializer only give rise to a modest speedup. On the other hand, when generating the compiler, `cogen` is appromimately three times faster.

In order to demonstrate that the relative speedup depends on the static input, here the program given to the "poolish-form" interpreter, we have specialized it to two other programs. The program `add` implements additions of $m + n$ by adding 1 to n, m times. The program `jump` is a loop containing 20 unconditional jumps. These two programs exploit different parts of then interpreter, and hence different speedups are obtained.

Program run	Run time		Code size	
	time	ratio	size	ratio
$[\![Int]\!](Add, 1, 1000000)$	24.1		123	
$[\![Int_{Add}]\!](1, 1000000)$	2.5	9.6	41	3
$[\![Int]\!](Jump, 1000000)$	29.6		123	
$[\![Int_{Jump}]\!](1000000)$	1.4	21.1	25	4.9

Notice how the speedup of `add` is 9.6 and of `jump` 21.1. The `primes` program containts several computations which cannot be done at specialization time. In `add`, there is only a few dynamic computations in the programs loop. Hence the larger speedup. In `jump`, in contrary, all the unconditional jumps are performed at specialization time, thus the speedup of 21.1. By adding more jumps to a program, an arbitrary large speedup can be achieved.

```
int spec_func_2_1 (Val * pstore)
{
   Val    *lstore, *store, *lval, res;
   int     pp0, rfunc, rpp, rpp1, rpp2;
   lstore = cmix_alloc_store
     (cmix_make_decl (VAR_DECL, "pow", 1, 0, cmix_make_dec (TYPE_INT),
      cmix_make_dec (VOID_DECL)));
   rpp = spec_init_code (1, "power", cmix_make_dec (TYPE_INT),
               cmix_make_decl (VAR_DECL, "n", 0, 0, cmix_make_dec (TYPE_INT),
               cmix_make_decl (VAR_DECL, "x", 1, 0, cmix_make_dec (TYPE_INT),
               cmix_make_dec (VOID_DECL))),
               cmix_make_decl (VAR_DECL, "pow", 1, 0, cmix_make_dec (TYPE_INT),
               cmix_make_dec (VOID_DECL)), pstore, gstore);
   rpp = spec_init_pending (pstore, lstore, gstore);
   while (spec_pending ()) {
         pp0 = spec_pend_pp ();        pstore = spec_pend_pstore ();
         lstore = spec_pend_lstore (); gstore = spec_pend_gstore ();
         rpp = spec_pend_processed (); rpp = spec_make_label (rpp);
         if (1 == pp0)  {
               rpp = spec_make_assign
                 (cmix_make_id_exp (BT_STATIC, 0, 2),
                  cmix__make_lift_exp
                  (cmix_make_ecall (BT_STATIC, "int_to_val",
                                    cmix_make_exps
                                    (cmix_make_int_exp (1),
                                     cmix_make_null_exp ()))));
               while (val_to_int (pstore[0])) {
                     rpp = spec_make_assign
                       (cmix_make_id_exp (BT_STATIC, 0, 2),
                        cmix_make_bop (BT_STATIC, 16,
                                       cmix_make_id_exp (BT_STATIC, 0, 2),
                                       cmix_make_id_exp (BT_STATIC, 0, 1)));
                     lval = &pstore[0];
                     lval[0] = int_to_val (val_to_int (pstore[0]) -
                                           val_to_int (int_to_val (1)));
                 }
               rpp = spec_make_endconf (pstore, gstore);
               rpp = spec_make_return (cmix_make_id_exp (BT_STATIC, 0, 2));
           }
         else {
               cmix_endconf = 1;
               return spec_error ("Program point not found");
           }
     }
   rpp = spec_close_pend ();
   cmix_endconf = 2;
   return spec_close_code ();
}
```

Figure 32: The structure of a power compiler (sugared)

```
Val sint_exec_fun_2_1 (Val* pstore)
{
      Val* lstore, *store, res, *lval;
      int foo;

      lstore = cmix_alloc_store
        (cmix_make_decl(VAR_DECL, "pow", 0, 0, cmix_make_dec(TYPE_INT),
         cmix_make_dec(VOID_DECL)));
      lval = &lstore[0];
      lval[0] = eval_exp(cmix_make_int_exp(1), pstore, lstore, gstore);
      lab5: if (val_to_int(eval_exp(cmix_make_id_exp(BT_STATIC, 0, 1),
                                    pstore, lstore, gstore))) {
           lval = &lstore[0];
           lval[0] = eval_exp(cmix_make_bop(BT_STATIC, 16,
                                cmix_make_id_exp(BT_STATIC, 0, 2),
                                cmix_make_id_exp(BT_STATIC, 1, 1)),
                              pstore, lstore, gstore);
           lval = &pstore[0];
           lval[0] = eval_exp(cmix_make_bop(BT_STATIC, 15,
                                cmix_make_id_exp(BT_STATIC, 0, 1),
                                cmix_make_int_exp(1)),
                              pstore, lstore, gstore);
           goto lab5;
        }
      else {
           cmix_endconf = 1;
           return eval_exp(cmix_make_id_exp(BT_STATIC, 0, 2),
                           pstore, lstore, gstore);
        }
}
```

Figure 33: Residual `power` *program without reduction of* `eval_exp()`

7.2.4 Specialization of a Self-Interpreter

In this section we consider specialization of the self-interpreter from Chapter 2, and in particular removal of a complete layer of interpretation. As example program we use `power`, and the goal is hence to obtain

$$[\![\texttt{mix}]\!]_C(\overline{\texttt{sint}}^{[Fun]}, \overline{\overline{\texttt{power}}^{[Fun]}}^{Val}) \approx \overline{\texttt{power}}^{[Fun]}$$

where `sint` is the self-interpreter.

In Figure 33 the result of specializing `sint` to `power` without reduction of the evaluation function `eval_exp()` is shown (we only shows the "main" function, and we have "pretty-printed" the program by hand).

```
Val sint_exec_fun_2_1 (Val* pstore)
{
      Val* lstore, *store, res, *lval;
      int foo;

      lstore = cmix_alloc_store
         (cmix_make_decl(VAR_DECL, "pow", 0, 0, cmix_make_dec(TYPE_INT),
          cmix_make_dec(VOID_DECL)));
      lval = &lstore[0];
      lval[0] = int_to_val(1);
      lab5: if (val_to_int(pstore[0])) {
         lval = &lstore[0];
         lval[0] = int_to_val(val_to_int(lstore[0]) * val_to_int(pstore[1]));
         lval = &pstore[0];
         lval[0] = int_to_val(val_to_int(pstore[0]) -
                              val_to_int(int_to_val(1)));
         goto lab5;
       }
      else {
             cmix_endconf = 1;
             return lstore[0];
       }
}
```

Figure 34: Residual `power` *program with reduction of* `eval_exp()`

The program is correct, but slow. A lot of compiler-generation time computation takes place at compile time, and memory is used for building syntax trees for expressions at compile time even though their structure are completely known at compiler generation time.

By turning on reduction of `eval_exp()` external function calls, as discussed in Section 3.4, the residual program given as Figure 34 results.

Observe that all the compiler generation time computation now has vanished. Furthermore, as discussed in Section 3.4, the `pstore`, and `lstore` can be split (*e.g.* in the unparsing process), and hence the original `power` program recovered modulo the tagging functions. Hence we conclude that the specializer is capable of specializing away a complete layer of interpretation.

The compiler generation time reduction of `eval_exp()` has an impack of the efficiency of the generated compilers. For example, consider the compiler generator `cogen` (a large compiler!). When a version of `cogen` generated without reduction of `eval_exp()` is applied to `spec` (reproducing the compiler generator), the run-time is 1.8 seconds. When the optimized version is applied, the run-time is 1.6 seconds.

7.3 Summary

In this chapter we have demonstrated our C-Mix system and provided benchmarks documenting the systems performance. We applied it to a general scanner, produced a `power` compiler, showed compiler generation for an interpreter, and demonstrated removal of a complete layer of interpretation.

The benchmarks are resonably good. Often an order of magnitude can be achieved by specializing an interpreter to a program, and running the compiler generator instead of applying `spec` is three times faster. The benchmarks have not, however, shown any significant improvement of stand-alone compilers. We believe this is due to the overhead in the compilers—initialization of pending and the code list—which dominates the run-time as long the programs are small.

Chapter 8

Conclusion

This chapter is organized as follows. In Section 8.1 we discuss related work and compared it to ours. We focus on the areas: program specialization of imperative languages, specialization of typed languages, binding time analysis and untagging analysis. We also give some more broad comparisons. In Section 8.2 we provide several directions for future work and the last section contains the conclusion over the thesis.

8.1 Related Work

In this section we related our work with previous and recent work. We mainly focus on specialization of imperative languages and applications, binding time analysis and dynamic typing which is strongly connected to our untagging analysis. We also provide a historical account of the development in the different areas.

8.1.1 Specialization of Imperative Languages

Historically, Ershov appears to be the first to consider program specialization of a (small) imperative language [Ershov 1977]. The specialization is formulated in terms of *mixed computation* but is not specified in detail. The underlying principle is the collection of program execution *traces* which correspond closely to the *poly-set*. Statements depending on unknown data are *suspended* while the others are executed. The *congruence principle* is formulated: a statement which depends on a suspended statement must itself be suspended.

The possibilities of compilation by specializing of an interpreter to a program are considered and explained, but compiler-generation by self-application is not mentioned.

Bulyonkov sets up a theoretical framework for *poly-variant* specialization of a similar small imperative language as Ershov's [Bulyonkov 1988]. Starting from a rather abstract notion of a programs expansion, more concrete algorithms for poly-variant specializations are developed. The overall structure of these are similar to those used in several implemented partial evaluators for functional languages.

Obstrovski reports a practical experiment in specialization of a parser generator written in an imperative language to a specific grammar [Ostrovski 1988]. Hereby a parser

tailored to a given grammar is obtained. No specific specialization algorithm is given, but various methods for improvement of the obtained code are discussed. This includes first and foremost rewriting of the program to be specialized; a technique which has been used extensively ever since the first program specializer saw the light of the day!

Meyer reports an *on-line* program specializer for a subset of the programming language Pascal [Meyer 1991]. Included is integer variables, procedures, assignments, `if`- and `while` statements and usual expressions. The work is a refinement of the pioneer work by Obstrovski [Ostrovski 1988]. Imperative data structures such as arrays and records are not treated—only base type variables.

The use of *on-line* determination of binding times makes the program specializer capable of doing more static computations than our *off-line* specializer may do. For example, a variable may initially be used statically, and get its status changed to *unknown* when *e.g.* assigned an unknown value. Furthermore, the on-line technique give by working principle poly-variant binding time assigments in contrary to ours mono-variant binding times. This, for instance, influences the specialization of functions. In our system, the binding time status of parameters are determined from all calls to it; in an on-line partial evaluator, the binding time assignment is typical solely depending on a concrete call.

In some respect the poly-variant specialization adopted by Meyer is less *poly-variant* than the technique we use. Consider as an example a residual `if` statement where a (known) variable v is assigned two different (known) values in the the two branches.

```
if ⟨dynamic test expression⟩
   then v := 87
   else v := 13
⟨Statements S⟩
```

Our technique unfoldt the statements S into the two branches giving maximal use of the static value of `v`. The specializer by Meyer specialize the two branches and observes that `v` is assigned two different values in the branches which makes specialization of S impossible. Hence, two *explicators* are inserted at the bottom of the two branches, one assigning `v` to 87, the other `v` to 13. Next, the binding time status of `v` is changed to "unknown" and S is specialized. The drawback of our method is that code duplication is likely to occur. We believe that *liveness* in most cases will reduce to code duplication to a tolerable extent. Furthermore, traditional flow-analyses may detect shared code in a subsequent pass [Aho *et al.* 1986].

A major lack in Meyer's approach is that in each function there exactly one specialization point—the first statement. Hence, a basic block will not be specialized to several values as in our specializer. This also means that treatment of `goto` is very hard—while it is the fundamental construct in our method.

For dealing with (global) side-effects in procedure calls a call-graph computation is used for *pre-annotating* such side-effects as *residual*. This correspond closely to our treatment of side-effects in (possible) recursive procedures, but we do allow some interprocedural side-effects to occur at specialization time.

Imperative data structures such as arrays and record are not considered. It should be straightforward to handle these using the same methods as developed in this thesis. In

Pascal, however, arrays are passed call-by-valye and not call-by-reference as in C. This ease splitting of partial static data structures since parameter passed variables can be split withour problems. Passing of call-by-refence variables reintroduce the problems, though.

Nirkhe and Pugh [Nirkhe and Pugh 1992] has developed an *off-line* non-self-applicable program specializer for a language similar to Meyer's. They use *dynamic scoping* instead of the usual static scoping, the advantages of this is unclear. Similar to Meyer, there is only one specialization point in each function, with the same consequences as described above. Non-local side-effects in functions are made dynamic. Treatment of in/out parameter passing is discussed but aliasing and sharing in general is not considered.

The paper contains a formal specification of the program specializer expressed in a denotational semantic style. The semantics is, however, cluttered up by all the tests for "compile-time" and "run-time" statements. Here our use of a two-level syntax is definitely more elegant and readable.

8.1.2 Self-applicable Program Specialization and Types

Bulyonkov and Ershov reports an *autoprojector* for a flow-chart language including assignments and simple flow-control commands [Bulyonkov and Ershov 1988]. The applied poly-variant specialization technique, as well as the autoprojector, is described in some details, but problems with representation of data structures *etc.* are not discussed. A generating extension for the power program is given, and its structure studied. The paper concludes with a discussion over the structure of generated compilers and traditionally "hand-written" ones. For instance, the necessity of splitting the activation stack in a specialized self-interpreter is discussed, as we did in Section 3.4. Many of the questions raised in the paper has later turned out to have nice solutions.

Barzdin describe a poly-variant specialization scheme for a small imperative language. The language include no functions, though, and self-application is not addressed in detail. A case-study shows the specialization of a Turing machine to a program. In contrast to most specializers from that time, Barzdin use *annotation* of statements into compile-time and run-time. This can be seen as a forerunner for two-level syntax, but these are not connected to our knowledge.

Jones provides a more theoretical approach to program specialization of an imperative flow-chart language [Jones 1988]. The stress is on *basic techniques* rather than advanced features. As an example, a simple program specializer is outlined and its application to a Turing machine interpreter is illustrated. The terms *program point*, *specialization point*, *program division*, and *congruent* division are formally defined. Several abstract algorithms are given. The last part of the paper deals with *termination* of program specialization.

Gomard and Jones describes a self-applicable partial evaluator for a small, untyped flow-chart language over the set of (Lisp) S-expressions [Gomard and Jones 1991a]. The basic techniques are the one decribed in [Jones 1988]. The language includes assignments, conditionals and unconditional jumps, but no functions and imperative data structures. The use of S-expressions eliminates the problem with self-representation of data structures and double encoding. Using the principles: basic block specialization, reduction and evaluation of expressions and transition compression, it is shown how compiler generation

by self-application can be done. The idea of *liveness* information for obtaining better sharing is introduced. However, since no variable splitting is done, the generated compilers are not as efficient and small as possible. For the same reason, the specializer cannot remove a complete layer of interpretation.

The first self-applicable partial evaluator for a strongly typed language is apparently the partial evaluator for a subset of LML by Launchbury [Launchbury 1991]. The representation problem is addressed, and a solution similar to the one we have used is described. An even more optimal (double) encoding of programs is developed. The idea is to use a *delayed* encoding of data structures—the double encoding is first done when the part of the data structure is needed. This reduces the heap usage dramatically. It is reported that the time for generating *cogen* is reduced from hours to minutes due to less garbage collection. The paper also addresses the problem of *untagging* a compiler generated by self-application.

Anne De Niel [De Niel *et al.* 1991] also addresses the data representation problem in self-interpreters for strongly typed languages. It is shown how the underlying implementation language (Lisp) can be used for reducing the data structures created. The technique can, of course, not be applied when the language in use is implemented directly on the computer by compilation, as C, and not interpreted.

A better understanding of the connection between types and partial evaluation is emerging. In [Jones 1990], Jones raises the question of *type safeness* of compilers generated by partial evaluation. The problem is: can it be assured that a compiler generated by self-application of a *correct* partial evaluator is *type-error* free if the interpreter is (type) correct? In particular, if the interpreter is written in a strongly typed language, can the interpreter be type checked and the type correctness used to guarantee that also the compiler is correct?

Preliminary results seems to indicate that it is possible to type a universal function in astrongly typed language. This has been further studied by Hagiya [Hagiya 1992]. We find this an interesting research area, and highly important in the further development of automatic compiler generation.

8.1.3 Compiler Generation

One of the initially driving force behind this project was to automate the methods of hand-crafting generating extensions accentuated by Pagan [Pagan 1988], [Pagan 1990].

The general idea is as follows:

1. Find a congruent separation of the binding times in the program (*i.e.* do binding time analysis)

2. Replace all residual statements with code generating statements (*i.e.* do the job of the specializer)

3. Apply some optimization techniques, *e.g.* replace a sequence of `if-else-if` statements with a `case` statements.

Only the last step requires some human effort and is hence not straightforward to automate. Automation of the two first steps for an imperative language is one of the results of this thesis.

We have implemented several of the examples given in [Pagan 1990], and used our specializer to generate specialized versions, *e.g.* specializing a scanner generator to a token specification for obtaining a language specific scanner. In must cases the code produced by our C-Mix is very similar to the code Pagan derives via his hand methods. Our system, of course, fails to recognize some possible optimizations which depends of the specific problem.

Heldal [Heldal 1991] has experimented with using partial evaluation for generation of compilers from higher-order functional languages to low-level machine code. The results are, however, very preliminary, and the method is hardly automated. We believe that using C as a portable "machine-language" is better than producing the machine code directly.

The MESS system, developed by Lee and Pleban, is a semantic-directed compiler generator system for automatic transformation of semantically specifications into executable compilers [Pleban and Lee 1988], [Lee 1989]. The semantics (and syntax) of the language is specified in a manner similar to *action semantics*, but in a syntax close to ML. The system supports the view that writing programming language specifications is closely connected with implementing an *interpreter* for it. When program specialization is used for compiler generation, *i.e.* language implementation, the basic view is that an interpreter is an *operational* description of the language.

The system give apparently good results, but even reasonably sized languages seems to require long specifications. It is reported that the specification of SOL/C, a language close to Core C, is 600 lines. Our self-interpreter for Core C is approximately 200 lines of C code which is definitely easier to write. On the other hand, our system is only capable of generating C code—the MESS system can generate all sorts of code.

8.1.4 Applications of Program Specialization

The applications of program specialization have mainly been into *compiler generation* and speeding-up *scientific computations*. The use of program specialization for compiler generation was described in previous section, and we hencefore concentrate on the second item.

Consel has applied his Schism partial evaluator to a subset of Algol 68, and obtained a compiler to Scheme [Consel and Danvy 1991]. It is shown how partial evaluation can solve part of the static semantics at compile-time, and is comparable to semantic-directed compiler generators. It is, however, often a more manageable task to write an interpreter than a formal specification, which is in favor of partial evaluation.

Berlin [Berlin and Weise 1990] has applied the FUSE on-line partial evaluator to several numerical analysis problems with good results. For example, it is reported that specialization of a generic n-body algorithm gave speed-ups of order 30–40. Many scientific problems is naturally expressed as an imperative, or iterative algorithm. We therefore find that our specializer may be extremely well-suited for further experiments in this area.

Application of partial evaluation in type inference is still unexploited [Hagiya 1992]. The idea is that in an interpreter for a strongly typed language, the type information is given *statically*. Therefore, by partial evaluating the interpreter, *type checking* can be done.

A major advantage with this approach is that *type checking* will benefit from stronger partial evaluators. It is in the paper shown how partial evaluation can handle some cases where ordinary type checking algorithms fails. This seems indeed very promising.

8.1.5 Binding Time Analysis

The introduction of the binding time analysis into program specialization originates back to the early Copenhagen Mix project [Jones *et al.* 1989]. By experiments it was found that compilers generated by self-application were both huge and slow. The reason was traced back to lack of compile time information. Consider self-application of an on-line *mix*:

$$[\![mix_1]\!](mix_2, int)$$

where *int* is an interpreter. In the mix_2, the code for reducing *e.g.* expressions in the input program is guarded by tests: "`if (static? exp) ...`". However, as the input values to *int* are unknown, these tests cannot be performed by mix_1. Hence, they appears in the compiler. By using binding time analysis, the tests *can* be performed, and therefore the compilers are reduced in size. New experiments with self-application of on-line partial evaluators have indicated that using clever programming the need for binding time analysis can be alleviated [Glück 1991], but it is still widely believed that binding time analysis is essential for efficient self-application of larger languages.

Originally, values was classified as being static or dynamic using abstract interpretation over the two-point domain $\{S, D\}$ [Jones *et al.* 1989]. The coarse descriptions of expressions have, however, several drawbacks. For instance, an alist representing the environment in an interpreter will be classified as completely dynamic (D) since parts of the list is dynamic. This is clearly unsatisfactory since then all (statically) lookup of variable names will not be performed at specialization time. The way to overcome these problems was by genius program transformations. For example, the environment list could be split into a name and a value list.

A refinement of the binding time analysis to describe *partial static structures* was developed by Mogensen [Mogensen 1988]. The binding time analysis was for a first-order version of Scheme, and can hence only describe partial static S-expressions.

The underlying principle use of *grammar-trees* for describing the staticness of a S-expression:

$$B ::= S \mid D \mid P(B, B)$$

where $P(x, x)$ describes a partial static cons-cell. In order to describe *e.g.* lists of arbitrary length, left-closed tree-grammars are used. If *types* are considered as *sets*, this framework correspond very closely to our use of binding time types for describing partial static (C-)structures.

For avoiding cluttering up the specializer with code for splitting partial static structures *during* the specialization, Mogensen developed a new analysis which, using *staging transformations* [Jørring and Sherlis 1986], automatically could strongly separate the binding times in a program [Mogensen 1989]. This resembles our requirement that a structure with both static and dynamic fields are split *before* the specialization phase. The result from the analysis was used to split functions which returns a partial structure into a complete static one and a dynamic one. An analysis for doing this is also given in the paper.

Binding time analysis based on *type inference* was founded by the Nielson's for inferring binding times in the two-level language TML [Nielson and Nielson 1988b]. The TML language is a general framework for semantically reasoning about languages, for instance abstract interpretation.

Gomard took a similar approach when he designed a binding time analysis for Lambda-Mix, a partial evaluator for a higher order version of the lambda calculus [Gomard 1990], [Gomard and Jones 1991b]. The used type system close resembles ours, but with functional types instead of array and structure types. Automatic insertion of the lift operator was not, however, incorporated into the algorithm.

For inferring the types, a backtracking version of the type inference algorithm W [Damas and Milner 1982] was constructed. When the normal algorithm W fails due to a unification failure, the modified algorithm returns an occurrence path to an operator which must be residual in order for the program to be *well-annotated*. Using Huet's fast unification algorithm, the binding time analysis run in $O(n^3)$, *i.e.* in polynomial time whereas abstract interpretation normally run in exponential time.

Using a subtyping inference algorithm with coercion [Fuh and Mishra 1989], Andersen and Mossin developed an automatic binding time analysis for a substantial subset of the Scheme language [Andersen and Mossin 1990]. *Coercion* allowed lifts of base values to dynamic, and hence solved the lift problem. The analysis was designed for use in the Similix higher order partial evaluator, but some design decisions in Similix made the analysis somewhat involved.

Henglein reformulated the binding time analysis for Lambda-Mix into the solution of a *constraint set* [Henglein 1991b]. By using fast find-union data structures with substitutions represented by equivalence classes, an algorithm running in almost linear time was obtained. The analysis is not, however, implemented and studied pragmatically.

A quite different approach to binding time analysis has been proposed by Hunt and Sands [Hunt and Sands 1991]. The idea is to use *partial equivalence relations* to describe the binding time "type" of expressions. The PER's gives automatically descriptions of partial static data structures in a manner similar to the type-based approach. We have no knowledge of an implementation of the binding time analysis, but we have a strong suspicion that it will be extremely inefficient.

8.1.6 Untagging Analysis and Dynamic Typing

The problem of *untagging* a program generated by specialization of a typed interpreter to a program, was first raised by Launchbury in his self-applicable partial evaluator for a

subset of LML [Launchbury 1991]. He did not, however, come up with a solution to the problem, but pointed out that if the user code up things in user defined data structures, then untagging is in general impossible to automate.

Our approach to untagging a program is strongly influenced by *dynamic typing*. The discipline *dynamic typing* originate back to the proposals of including a *Dynamic* type in strongly typed languages [Abadi *et al.* 1989]. The idea is to combine the benefits from strongly-typed languages with the user convenience in dynamically typed programs. For example, in strongly typed languages, list with elements of different type is normally not possible, but in some user applications it may be convenient to allow these. With dynamic typing, the user can in some places "turn off" the statically typing, and do defer the type checking to run-time.

Thatte developed a type inference algorithm for a generalized Hindley-Milner type system with dynamic typing [Thatte 1988]. The algorithm is a subtyping extension to W with coercions. O'Keefe later showed that Thattes semi-decidable typing problem is decidable, but gave no algorithm [O'Keefe and Wand 1991].

Gomard used the term *partial type inference* in another meaning than Thatte. Here the goal was: given a program in a completely untyped language to infer as much static type information as possible. As a side-effect, the dynamically typed parts of the program was annotated and should hence be surrounded with run-time type checks. For inferring the partial type information, a backtracking version of algorithm W was outlined [Gomard 1990].

Henglein adapted his binding time analysis and obtained a constraint set based algorithm for inferring partial types [Henglein 1991a]. The untagging analysis described in Chapter 6 has to some extend been inspired by the work of Henglein. A prototype implementation of the algorithm in Scheme has given very promising results. In some cases up to 80 % of the tagging operations can be eliminated safely [Henglein 1991c].

Shivers has independently developed a type recovery analysis for Scheme, using a CPS representations where control-flow is explicit [Shivers 1991]. The basic principle is abstract interpretation but the outcome of the different analyses appears to be similar.

8.2 Future Research

In this section we discuss future work and give several directions for continuation of this work. We concentrate mainly on the issues: treatment of a larger subset of C, especially pointers and heap allocated data structures, efficient static program analyses and their use in program specialization, and applications for program specialization.

8.2.1 Specialization of C

In this thesis we have described specialization of a large subset of C. Almost all kind of statements and expressions can be handled. The missing statements and expressions, *e.g.* increment of variables, can easily be handled by a priori program transformations. The major problem to overcome is treatment of C's data structures. The involved semantics of C, for example casts, void pointers, negative array indexing, function pointers *etc* make

treatment of the full language very hard. We believe, however, heap allocated structures and pointers to these are managable to implements in a semantically safe way.

There are two major places to attack: the strenght of the specializer and stronger static analyses supplying the specializer with more informations such as side effects and aliasing. We discuss static program analyses in the next section.

In the following we describe some initial solutions of the problem of handling heap-allocated structures. A simple solution is the very conservative treatment: classify a data structure completely static or dynamic. This leaves the problem to the binding time analysis, but here an alias analysis can overcome the problems. This is not, however, satisfactory in many applications. What is needed is an extended binding time assignment which allows partially static data structures. Consider for instance a list where each node consist of a data field and a "next" pointer. Here, the pointer field must be both static and dynamic; it is needed at both specialization time and run time. Similarly holds for *e.g.* binary trees, syntax trees and so on. This indicate that the binding time analysis somehow must be parametrized with the user defined data types.

Our first requirement is that the treated C subset is strongly typed ruling out casts. Hence, a pointer can only point to objects of the type they are declared to. Next, we will be content with a mono-variant binding time description of structures. Hence, in *e.g.* a list, all the elements must possess the same binding time.

For doing memory comparisons and copying, a tagged representation can be used. It is important that circularities is detected; otherwise the specializer might loop while comparing the specialization time stores!

The major problem is the binding time analysis. We believe a combination of an alias analysis and a parameterization with data types may be sufficient, but we have not investigated this further.

8.2.2 Efficient Program Analyses

The C-Mix differ from the must other partial evaluators for imperative languages in the use of static program analyses for providing the specializer with a maximum of compile time information. The foremost analysis is the binding time analysis, but we believe the fruitful way to handling more complicated data structures is through static analyses and possible some a priori program transformations.

It is often relatively easy to express the properties one wish to infer in a type inference system—and it is even elegant! Further, is it usually pretty straightforward to develop an algorithm implementing the analysis by hacking one of the standard type inference algorithms, *e.g.* algorithm W. Furthermore, even more interestingly, a large part of the theory behind type inference may be adapted to show the correctness of the new analyses. For example, Milner's theorem may be reformulated into: "Well-analysed programs do not whatever".

However, the problem with this technique is *efficiency*. Typically, an analysis based on abstract interpretation will be faster even though the type inference in theory should possess better run-time behavior. For example, abstract interpretation normally run in exponential time, whereas a type inference algorithm using fast unification run in

polynomial time. Implementations of type checking in known to behave very well even though they theoretically have intractable run time characteristic. To our knowledge, the same experiences seems not to dominate other uses of type inference in static program analyses, for example *strictness analysis* [Hall 1991]. The constraint set solving method used in this project may be applicable in these areas too.

An important extension to the binding time analysis from this thesis is poly-variance. Poly-variance can be introduced in several steps. A "poor-man" poly-variance would be to have a division for each program point instead of a common division for the whole function. Consider the following program fragment.

```
int foobar(int l)
{
    int m;
    1: assign m = l
    2: assign m = 3
    3: return m
}
```

Suppose `l` is dynamic. A mono-variant, congruent division would claasify `m` as dynamic as it is assigned a dynamic value at program point 1. However, clearly at program point 2 and 3, `m` is static. A pointwise division could yield better division. When the binding time status of variables is changes from static to dynamic, for example in the `power` function, explicators can be inserted, possibly before specialization time.

The next step is a "real" poly-variant division. With the view of binding time analysis as type checking in the two-level language, poly-variance is tighly connected with polymorphism in programming languages. It should be easy to extend the binding time analysis to handle this. The constraint solving algorithm must be modified, though. We believe the best approach to introduce poly-variance is by explicit copying of poly-variant functions such that an instance exist for every needed division. This way the specializer developed in this thesis can be applied without modifications.

8.2.3 Applications of Program Specialization

In our opinion, there is a pressing need for several practical experiments with program specializers. Especially, application to "real-scale" problems should be investigated further.

Preliminary experiments of partial evaluating scientific hard real-time problems clearly show that large speed-ups can be achieved [Berlin and Weise 1990]. We believe that a large number of applications can be found in the fields of numerical analyses and combinatorial problems.

Several other application areas exists, however. For example, in many novel programming languages, *modularity* is a central concept for structural programming. The idea is that a program is built as a set of modules with well-defined interfaces. This way a user may use a module without any other knowledge of the module but its interface. The modularity causes penalties in the implementation: the repeated call of interface functions

is time-wasting and data is likely to be packed into a general data structures. Program specialization may be able to remove many of these superfluous function calls and the packing of data values.

Since our program specializer is self-applicable, it is natural to take a closer look at compiler generation by partial evaluation. A nice feature with our system is that it generates efficient code. However, it is normally more tractable to write an interpreter in a functional language due to the data structures, the more "clean" use of functions *etc.* An obvious solution would be to write an interpreter (in C) for a functional language and when use the program specializer to compile with.

8.3 Conclusion

We have addressed the problem of *self-applicable program specialization* of a realistic, imperative programming language: a substantial subset of C. We considered handling of a language with effects and mutual states, and imperative data structures.

The program specializer was formally defined via a two-level operational semantics where additional requirements were formulated in terms of *well-annotatedness*. We gave algorithms which implements the two-level semantics.

The binding time analysis was formulated in the setting of type inference, and an efficient implementation can be done by the means of constraint solving. A similar technique was used in the development of an *untagging analysis*.

Major part of the C program specializer has been implemented; the binding time and untaggin analysis remain unimplemented, though. We provided several experimental results, using the C-Mix system to specialize general programs, and automatic compiler generation via self-applications.

We consider automatic program specialization of C as an area with many potential application possibilities. Further research towards stronger and more powerful C specializer should therefore definitely be undertaken.

Bibliography

[Abadi *et al.* 1989] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin, Dynamic typing in a statically-typed language, in *Proc. of 16th. Annual ACM Symp. on Principles of Programming Languages,* pages 213–227, ACM, 1989.

[Aho *et al.* 1986] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison Wesley, 1986.

[Andersen 1991a] L.O. Andersen, *C Program Specialization,* Master's thesis, DIKU, University of Copenhagen, Denmark, December 1991. DIKU Student Project 91-12-17, 134 pages.

[Andersen 1991b] L.O. Andersen, Correctness Proof for a Self-Interpreter, December 1991. DIKU Student Project 91-12-16, 21 pages. DIKU, University of Copenhagen.

[Andersen 1992a] L.O. Andersen, Partial Evaluation of C and Automatic Compiler Generation (extended abstract), in *Proc. International Workshop on Complier Construction,* 1992.

[Andersen 1992b] L.O. Andersen, Self-Applicable C Program Specialization, in *Proceeding of PEPM'92: Partial Evaluation and Semantics-Based Program Manipulation,* 1992.

[Andersen and Gomard 1992] L.O. Andersen and C.K. Gomard, Speedup Analysis in Partial Evaluation: Preliminary results, in *Proc. of Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'92),* 1992.

[Andersen and Mossin 1990] L.O. Andersen and C. Mossin, Binding Time Analysis via Type Inference, October 1990. DIKU Student Project 90-10-12, 100 pages. DIKU, University of Copenhagen.

[Barzdin 1988] G. Barzdin, Mixed Computation and Compiler Basis, in *Partial Evaluation and Mixed Computation,* edited by D. Bjørner, A.P. Ershov, and N.D. Jones, pages 15–26, North-Holland, 1988.

[Beckman *et al.* 1976] L. Beckman *et al.,* A Partial Evaluator, and Its Use as a Programming Tool, *Artificial Intelligence* 7,4 (1976) 319–357.

[Berlin and Weise 1990] A. Berlin and D. Weise, Compiling Scientific Code Using Partial Evaluation, *IEEE Computer* 23,12 (December 1990) 25–37.

[Bondorf 1990a] A. Bondorf, Automatic autoprojection of higher order recursive equations, in *ESOP'90, 3rd European Symposium on Programming, Copenhagen, Denmark. Lecture Notes in Computer Science 432,* edited by Neil D. Jones, pages 70–87, Springer Verlag, May 1990.

[Bondorf 1990b] A. Bondorf, *Self-Applicable Partial Evaluation,* PhD thesis, DIKU, University of Copenhagen, 1990.

[Bondorf 1992] Anders Bondorf, Improving binding times without explicit cps-conversion, in *1992 ACM Conference on Lisp and Functional Languages. San Francisco, California,* June 1992. (To appear).

[Bondorf and Danvy 1990] A. Bondorf and O. Danvy, *Automatic Autoprojection of Recursive Equations with Global Variables and Abstract Data Types,* Technical Report 90/4, DIKU, University of Copenhagen, Denmark, 1990.

[Bondorf *et al.* 1988] A. Bondorf, N.D. Jones, T. Mogensen, and P. Sestoft, *Binding Time Analysis and the Taming of Self-Application,* Draft, 18 pages, DIKU, University of Copenhagen, Denmark, August 1988.

[Bourdoncle 1990] F. Bourdoncle, Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity, in *PLILP'90,* pages 307–323, Springer Verlag, 1990.

[Bulyonkov 1988] M.A. Bulyonkov, A Theoretical Approach to Polyvariant Mixed Computation, in *Partial Evaluation and Mixed Computation,* edited by D. Bjørner, A.P. Ershov, and N.D. Jones, pages 51–64, North-Holland, 1988.

[Bulyonkov and Ershov 1988] M.A. Bulyonkov and A.P. Ershov, How Do Ad-Hoc Compiler Constructs Appear in Universal Mixed Computation Processes?, in *Partial Evaluation and Mixed Computation,* edited by D. Bjørner, A.P. Ershov, and N.D. Jones, pages 65–81, North-Holland, 1988.

[Consel 1988] C. Consel, New Insights into Partial Evaluation: The Schism Experiment, in *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 1988. (Lecture Notes in Computer Science, vol. 300),* edited by H. Ganzinger, pages 236–246, Springer Verlag, 1988.

[Consel 1990] C. Consel, Binding Time Analysis for Higher Order Untyped Functional Languages, in *1990 ACM Conference on Lisp and Functional Programming, Nice, France,* pages 264–272, ACM, 1990.

[Consel and Danvy 1991] C. Consel and O. Danvy, Static and Dynamic Semantics Processing, in *Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida,* pages 14–24, ACM, January 1991.

[Damas and Milner 1982] L. Damas and R. Milner, Principal type-schemes for functional programs, in *Ninth ACM Symposium on Principles of Programming Languages,* pages 207–212, ACM, 1982.

[De Niel *et al.* 1991] A. De Niel, E. Bevers, and K. De Vlaminck, Partial Evaluation of Polymorphically Typed Functional Languages: The Representation Problem, in *Analyse Statique en Programmation Équationnelle, Fonctionnelle, et Logique, Bordeaux, France, Octobre 1991. (Bigre, vol. 74),* edited by M. Billaud *et al.,* pages 90–97, IRISA, Rennes, France, 1991.

[Ershov 1977] A.P. Ershov, On the Partial Computation Principle, *Information Processing Letters* 6,2 (April 1977) 38–41.

[Fuh and Mishra 1989] Y. Fuh and P. Mishra, Polymorphic Subtype Inference: Closing the Theory-Practice Gap, in *TAPSOFT (LNCS 352),* edited by J. Díaz and F. Orejas, pages 167–183, Springer Verlag, 1989.

[Futamura 1971] Y. Futamura, Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler, *Systems, Computers, Controls* 2,5 (1971) 45–50.

[Glück 1991] R. Glück, Towards Multiple Self-Application, in *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991),* pages 309–320, ACM, 1991.

[Gomard 1990] C.K. Gomard, Partial Type Inference for Untyped Functional Programs, in *1990 ACM Conference on Lisp and Functional Programming, Nice, France,* pages 282–287, ACM, 1990.

[Gomard 1992] C. K. Gomard, A Self-applicable Partial Evaluator for the Lambda Calculus: Correctness and Pragmatics, *ACM Transactions on Programming Languages and Systems* (1992). (to appear).

[Gomard and Jones 1991a] C.K. Gomard and N.D. Jones, Compiler Generation by Partial Evaluation: a Case Study, *Structured Programming* 12 (1991) 123–144.

[Gomard and Jones 1991b] C.K. Gomard and N.D. Jones, A Partial Evaluator for the Untyped Lambda-Calculus, *Journal of Functional Programming* 1,1 (January 1991) 21–69.

[Hagiya 1992] M. Hagiya, An Operational Approach to Type Safety, *submitted to LICS'92* (1992).

[Hall 1991] C. Hall, How to Persuade a Typechecker to do Strictness Analysis, in *Fourth Annual Glasgow Workshop on Functional Programming,* pages 151–164, 1991.

[Haraldsson 1978] A. Haraldsson, A Partial Evaluator, and Its Use for Compiling Iterative Statements in Lisp, in *Fifth ACM Symposium on Principles of Programming Languages, Tucson, Arizona,* pages 195–202, 1978.

[Heldal 1991] R. Heldal, Generating more practical Compilers by Partial Evaluation, in *Proc. of Workshop of Functional Programming Languages,* 1991.

[Henglein 1991a] F. Henglein, Dynamic Typing, *(Submitted to ESOP'92)* (1991).

[Henglein 1991b] F. Henglein, Efficient Type Inference for Higher-Order Binding-Time Analysis, in *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523),* edited by J. Hughes, pages 448–472, ACM, Springer Verlag, 1991.

[Henglein 1991c] F. Henglein, Global tagging optimization by type inference, *(submitted to LFP-'92)* (1991).

[Holst 1991] C.K. Holst, Finiteness Analysis, in *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523),* edited by J. Hughes, pages 473–495, ACM, Springer Verlag, 1991.

[Hunt and Sands 1991] S. Hunt and D. Sands, Binding Time Analysis: A New PERspective, in *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991),* pages 154–165, ACM, 1991.

[Jones 1988] N.D. Jones, Automatic Program Specialization: A Re-Examination from Basic Principles, in *Partial Evaluation and Mixed Computation,* edited by D. Bjørner, A.P. Ershov, and N.D. Jones, pages 225–282, North-Holland, 1988.

[Jones 1990] N.D. Jones, Partial Evaluation, Self-Application and Types, in *Automata, Languages and Programming. 17th International Colloquium, Warwick, England. (Lecture Notes in Computer Science, vol. 443),* edited by M.S. Paterson, pages 639–659, Springer Verlag, 1990.

[Jones *et al.* 1989] N.D. Jones, P. Sestoft, and H. Søndergaard, Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation, *Lisp and Symbolic Computation* 2,1 (1989) 9–50.

[Jørring and Sherlis 1986] U. Jørring and W.L. Sherlis, Compilers and Staging Transformation, in *Proc. Thirteenth ACM POPL Sympl,* pages 86–96, 1986.

[Kahn 1987] G. Kahn, *Natural Semantics,* Rapports de Recherche, Institut National de Recherche en Informatique et en Automatique, Febuary 1987.

[Kernighan and Ritchie 1988] B.W. Kernighan and D.M. Ritchie, *The C programming language (Draft-Proposed ANSI C), Software Series,* Prentice-Hall, second edition edition, 1988.

[Launchbury 1990] J. Launchbury, *Projection Factorisations in Partial Evaluation,* PhD thesis, Dep. of Computing Science, University of Glasgow, Glasgow G12 8QQ, 1990.

[Launchbury 1991] J. Launchbury, A Strongly-Typed Self-Applicable Partial Evaluator, in *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523),* edited by J. Hughes, pages 145–164, ACM, Springer Verlag, 1991.

[Lee 1989] P. Lee, *Realistic Compiler Generation,* The MIT Press, 1989.

[Meyer 1991] U. Meyer, Techniques for Partial Evaluation of Imperative Languages, in *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991),* pages 94–105, ACM, 1991.

[Milner 1978] R. Milner, A theory of type polymorphism in programming, *Journal of Computer and Systems Sciences* 17 (1978) 348–375.

[Mogensen 1986] T. Mogensen, *The Application of Partial Evaluation to Ray-Tracing,* Master's thesis, DIKU, University of Copenhagen, Denmark, 1986.

[Mogensen 1988] T.Æ. Mogensen, Partially Static Structures in a Self-Applicable Partial Evaluator, in *Partial Evaluation and Mixed Computation,* edited by D. Bjørner, A.P. Ershov, and N.D. Jones, pages 325–347, North-Holland, 1988.

[Mogensen 1989] T.Æ. Mogensen, Separating Binding Times in Language Specifications, in *Fourth International Conference on Functional Programming Languages and Computer Architecture, London, England, September 1989,* pages 14–25, ACM Press and Addison-Wesley, 1989.

[Nielson and Nielson 1988a] F. Nielson and H. R. Nielson, *The TML-Approach to Compiler-Compilers,* Technical Report 1988-47, Department of Computer Science, Technical University of Denmark, 1988.

[Nielson and Nielson 1988b] H.R. Nielson and F. Nielson, Automatic Binding Time Analysis for a Typed λ-Calculus, *Science of Computer Programming* 10 (1988) 139–176.

[Nirkhe and Pugh 1992] V. Nirkhe and W. Pugh, Partial Evaluation and High-Level Imperative Programming Languages with Applications in Hard Real-Time Systems, in *Nineteenth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1992,* ACM, 1992.

[O'Keefe and Wand 1991] Patrick O'Keefe and Mitchel Wand, Type Inference for Partial Types is Decidable, 1991. 11 pages.

[Ostrovski 1988] B.N. Ostrovski, Implementation of Controlled Mixed Computation in System for Automatic Development of Language-Oriented Parsers, in *Partial Evaluation and Mixed Computation,* edited by D. Bjørner, A.P. Ershov, and N.D. Jones, pages 385–403, North-Holland, 1988.

[Pagan 1988] F.G. Pagan, Converting Interpreters into Compilers, *Software — Practice and Experience* 18,6 (June 1988) 509–527.

[Pagan 1990] F.G. Pagan, *Partial Computation and the Construction of Language Processors,* Prentice-Hall, 1990.

[Peyton Jones 1991] S.L. Peyton Jones, The Spineless Tagless G-machine: A Second Attempt, in *Draft Proceedings, Fourth Annual Workshop on Functional Programming, Isle of Skye, Scotland, August 1991,* pages 438–484, Department of Computing Science, University of Glasgow, Scotland, 1991.

[Pleban and Lee 1988] U.F. Pleban and P. Lee, An Automatically Generated, Realistic Compiler for an Imperative Programming Language, in *Proc. of the SIGPLAN '88 Conf. on Programming Language Design and Implementation,* pages 222–232, ACM, 1988.

[Plotkin 1981] G. Plotkin, *A Structural Approach To Operational Semantics,* Technical Report DAIMI FN-19, Computer Science Department, AArhus University, Denmark, Ny Munkegade, DK 8000 Aarhus C, Denmark, 1981. Technical Report.

[Robinson 1965] J. A. Robinson, A Machine-oriented Logic Based on the Resolution Principle, *Journal of the ACM* 12 (1965) 23–41.

[Romanenko 1988] S.A. Romanenko, A Compiler Generator Produced by a Self-Applicable Specializer Can Have a Surprisingly Natural and Understandable Structure, in *Partial Evaluation and Mixed Computation,* edited by D. Bjørner, A.P. Ershov, and N.D. Jones, pages 445–463, North-Holland, 1988.

[Romanenko 1990] S.A. Romanenko, Arity Raiser and Its Use in Program Specialization, in *ESOP '90, 3rd European Symposium on Programming,* edited by N. Jones, pages 341–360, Keldysh Institute of Applied Mathematics, May 1990.

[Shivers 1991] O. Shivers, *Control-Flow Analysis of Higher-Order Languages or Taming Lambda,* PhD thesis, Carnegie Mellon University, may 1991.

[Stallman 1991] R.M. Stallman, *Using and Porting Gnu CC,* Free Software Foundation Inc, 675 Mass Ave, Cambridge, 1.39 edition, january 1991.

[Thatte 1988] Satish Thatte, Type Inference with Partial Types, *Proc. ICALP 88* (1988) 615–629.

[Weise *et al.* 1991] D. Weise, R. Conybeare, E. Ruf, and S. Seligman, Automatic Online Partial Evaluation, in *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523),* edited by J. Hughes, pages 165–191, ACM, Springer Verlag, 1991.

Appendix A

The Core C Self-Interpreter

```
/*
 *
 *      File:     sint.c
 *      Author:   Lars Ole Andersen (lars@diku.dk)
 *      Created:  Mon Jun  1 05:36:19 1992
 *      Modified: Mon Jun  1 05:36:19 1992
 *      Content:  Core C Self-Interpreter
 *
 */

#ifndef _CMIX
#include <stdio.h>
#endif
#include "memory.h"
#include "asyntax.h"
#include "value.h"
#include "expr.h"

extern sint_error(char *);

Fun
  *pgm;                                 /* Program                      */
Val
  *gstore;                              /* Global store                 */


Val
  sint(Fun *, Val *),                   /* Execute program              */
  sint_exec_fun(int, Val *);            /* Execute function             */

Val
```

```
sint(Fun *program, Val *parameter)
{
    Val foo;

    /* Initialize                                                    */
    pgm = program;                      /* Store program in global var     */
    gstore = cmix_alloc_store(cmix_global_decl(pgm[0]));
                                        /* Allocate global store           */

    /* Execute main function                                         */
    foo = sint_exec_fun(MAIN_FUNC, parameter);

    /* That's it                                                     */
    return foo;
}


/* sint_exec_fun: execute function                                   */
Val
sint_exec_fun(int func, Val *pstore)
{
    int
      pp,                               /* Pending index                   */
      n,
      foo;

    Val
      *lstore,                          /* Local store                     */
      *store,                           /* Parameter store for calls       */
      res,                              /* Result temporary                */
      *lval;                            /* Left value pointer              */


    /* Initialization:
     * Allocate store for local variables,
     */
    lstore = cmix_alloc_store(cmix_local_decl(pgm[func]));
                                        /* Allocate local store    */
    pp = 1;

    /* Standard execution                                            */
    while (pp)
      {
          switch (cmix__stmt_kind(pgm[func], pp))
```

```
{
case ASSIGN_STMT:
    /*
     * Assignments:
     * Evaluate the expressions and perform an assignment
     */
    switch (cmix__assign_loc(pgm[func], pp))
      {
      case PARAMETER_VAR:
          lval = &pstore[cmix__assign_var(pgm[func], pp)];
          break;

      case LOCAL_VAR:
          lval = &lstore[cmix__assign_var(pgm[func], pp)];
          break;

      case GLOBAL_VAR:
          lval = &gstore[cmix__assign_var(pgm[func], pp)];
          break;
      }
    for (n = 0; n < cmix__assign_nsindex(pgm[func], pp);
         n += 1)
      lval = &(val_to_ptr(lval[0])
               [val_to_int
                (eval_exp
                 (cmix__assign_index(pgm[func],pp, n),
                  pstore, lstore, gstore))]);
    assign(lval[0],eval_exp(cmix__assign_exp(pgm[func], pp),
                            pstore, lstore, gstore));
    pp += 1;
    break;

case GOTO_STMT:
    /*
     * Goto:
     * Perform transition compression: jump to the target
     */
    pp = cmix__goto_label(pgm[func], pp);
    break;

case IF_STMT:
    /*
     * If:
     * Evaluate the test-expression and perform
```

```
     * transition compression
     */
    if (val_to_int(eval_exp(cmix__if_exp(pgm[func], pp),
                            pstore, lstore, gstore)))
      pp = cmix__if_then(pgm[func], pp);
    else
      pp = cmix__if_else(pgm[func], pp);
    break;

case CALL_STMT:
    /*
     * Call statement:
     * Allocate store for arguments and evaluate these.
     * Execute the function and update the local store
     * with the result
     */
    store = cmix_alloc_val(cmix__call_nspars(pgm[func],pp));
    for (n = 0; n < cmix__call_nspars(pgm[func], pp); n += 1)
      assign(store[n],
             eval_exp(cmix__call_spar(pgm[func], pp, n),
                      pstore, lstore, gstore));
    assign(res,
           sint_exec_fun(cmix__call_fun(pgm[func], pp), store));
    foo = cmix_dealloc_store(store);
    switch (cmix__call_loc(pgm[func], pp))
      {
      case PARAMETER_VAR:
          lval = &pstore[cmix__call_var(pgm[func], pp)];
          break;
      case LOCAL_VAR:
          lval = &lstore[cmix__call_var(pgm[func], pp)];
          break;
      case GLOBAL_VAR:
          lval = &gstore[cmix__call_var(pgm[func], pp)];
          break;
      }
    assign(lval[0], res);
    pp += 1;
    break;

case RETURN_STMT:
    /*
     * Return:
     * Reduce the expression and built residual statement
```

```
                 */
                return eval_exp(cmix__return_exp(pgm[func], pp),
                                pstore, lstore, gstore);
                break;

            default:
                sint_error("unknown kind of syntax\n");

            }
      }
    return int_to_val(0);
}
```

Appendix B

The Core C Program Specializer

```
/*
 *
 *      File:     spec.c
 *      Author:   Lars Ole Andersen (lars@diku.dk)
 *      Created:  Wed Apr 22 13:52:16 1992
 *      Modified: Fri May 29 14:41:52 1992
 *      Content:  C-Mix Specializer (version 3)
 *
 */

#ifndef _CMIX
#include <stdio.h>
#endif
#include "asyntax.h"
#include "value.h"
#include "expr.h"
#include "memory.h"
#include "pend.h"
#include "codegen.h"
#include "cmix.h"
#define DECLARE
#include "spec.h"

Fun
  *pgm;                                 /* Program                      */
Val
  *gstore;                              /* Global store                 */

int
  spec_func(int, Val *);                /* Specialize function          */
Val
```

```
  spec_exec_fun(int, Val *);            /* Execute function               */



Fun
*specialize(Fun *program, Val *parameter)
{
    int foo;

    /* Initialize                                                         */
    pgm = program;                      /* Store program in global var    */
    gstore = cmix_alloc_store(cmix_global_decl(pgm[0]));
                                        /* Allocate global store          */
    foo = spec_init_global(cmix_global_decl(pgm[0]));

    /* Specialize the MAIN function to static parameters                  */
    foo = spec_func(MAIN_FUNC, parameter);

    /* That's it                                                          */
    return spec_get_pgm();
}


/************************************************************************
 * Function specialization                                              *
 ************************************************************************/

int
spec_func(int func, Val *pstore)
{
    int
      pp, pp0,                          /* Program index                  */
      n,
      rfunc,                            /* Residual function index        */
      rpp, rpp1, rpp2;                  /* Residual program points        */

    Val
      *lstore,                          /* Local store                    */
      *store,                           /* Parameter store for calls      */
      res,                              /* Result temporary               */
      *lval;                            /* Left value pointer             */

    /* Initialization:
     * Allocate store for local variables,
```

```
     * initialize code generation and pending list handling,
     *  and insert first program point into pending.
     */
#ifdef DEBUG_SPEC
    fprintf(stderr, "DEBUG_SPEC: Specializer function %d\n", func);
#endif
    lstore = cmix_alloc_store(cmix_local_decl(pgm[func]));
    rpp = spec_init_code(func,
                         cmix_fun_name(pgm[func]),
                         cmix_fun_type(pgm[func]),
                         cmix_param_decl(pgm[func]),
                         cmix_local_decl(pgm[func]),
                         pstore, gstore);
    rpp = spec_init_pending(pstore, lstore, gstore);
    goto pending_loop;

    /* Pending loop                                                    */
 pending_loop:
    while (spec_pending())
      {
          pp0 = spec_pend_pp();
          pstore = spec_pend_pstore();
          lstore = spec_pend_lstore();
          gstore = spec_pend_gstore();
          rpp = spec_pend_processed();
          rpp = spec_make_label(rpp);

          /* Lookup the specialization point under static control      */
          for (pp = 1; pp <= cmix_num_stmt(pgm[func]); )
            {
                if (pp == pp0)
                  goto exec;
                pp += 1;
                while (!spec_point(pp, pgm[func]))
                  pp += 1;
            }
          return spec_error("Program point not found");

          /* Two-level execution                                       */
      exec:  while (pp)
              {
#ifdef DEBUG_SPEC
                fprintf(stderr, "DEBUG SPEC "
                        "Two-level execution of %d in %d\n", pp, func);
```

```
#endif
                switch (cmix__stmt_kind(pgm[func], pp))
                  {
                  case ASSIGN_STMT:
                      /*
                       * Static assignments:
                       * Evaluate the expressions and perform an assignment
                       */
                      switch (cmix__assign_loc(pgm[func], pp))
                        {
                        case PARAMETER_VAR:
                            lval = &pstore[cmix__assign_var(pgm[func], pp)];
                            break;

                        case LOCAL_VAR:
                            lval = &lstore[cmix__assign_var(pgm[func], pp)];
                            break;

                        case GLOBAL_VAR:
                            lval = &gstore[cmix__assign_var(pgm[func], pp)];
                            break;
                        }
                      for (n = 0; n < cmix__assign_nsindex(pgm[func], pp);
                           n += 1)
                        lval = &(val_to_ptr(lval[0])
                                 [val_to_int
                                  (eval_exp
                                   (cmix__assign_index(pgm[func],pp, n),
                                    pstore, lstore, gstore))]);
                      assign(lval[0],eval_exp(cmix__assign_exp(pgm[func], pp),
                                              pstore, lstore, gstore));
                      pp += 1;
                      break;

                  case _ASSIGN_STMT:
                      /*
                       * Dynamic assignment:
                       * Reduce the expressions and built a residual statement
                       */
                      rpp = spec_make_assign
                        (reduce_exp(cmix__assign_lexp(pgm[func], pp),
                                    pstore, lstore, gstore),
                         reduce_exp(cmix__assign_exp(pgm[func], pp),
                                    pstore, lstore, gstore));
```

```
    pp += 1;
    break;

case GOTO_STMT:
    /*
     * Static goto:
     * Perform transition compression: jump to the target
     */
    pp = cmix__goto_label(pgm[func], pp);
    break;

case _GOTO_STMT:
    /*
     * Dynamic goto:
     * Generate residual goto and insert specialization
     * point into pending if not already processed
     */
    rpp = spec_pend_insert(cmix__goto_label(pgm[func], pp),
                           cmix__goto_pdead(pgm[func], pp),
                           cmix__goto_ldead(pgm[func], pp),
                           cmix__goto_gdead(pgm[func], pp),
                           pstore, lstore, gstore);
    rpp = spec_make_goto(rpp);
    pp = 0;
    break;

case IF_STMT:
    /*
     * Static if:
     * Evaluate the test-expression and perform
     * transition compression
     */
    goto IF; /* Force next if to be specialized */
IF: if (val_to_int(eval_exp(cmix__if_exp(pgm[func], pp),
                            pstore, lstore, gstore)))
      pp = cmix__if_then(pgm[func], pp);
    else
      pp = cmix__if_else(pgm[func], pp);
    break;

case _IF_STMT:
    /*
     * Dynamic if:
     * Reduce expression and built residual if.  Specialize
```

```
         * both then- and else-branches
         */
        rpp2 = spec_pend_insert(cmix__if_else(pgm[func], pp),
                                cmix__ife_pdead(pgm[func], pp),
                                cmix__ife_ldead(pgm[func], pp),
                                cmix__ife_gdead(pgm[func], pp),
                                pstore, lstore, gstore);
        rpp1 = spec_pend_insert(cmix__if_then(pgm[func], pp),
                                cmix__ift_pdead(pgm[func], pp),
                                cmix__ift_ldead(pgm[func], pp),
                                cmix__ift_gdead(pgm[func], pp),
                                pstore, lstore, gstore);
        rpp =
          spec_make_if(reduce_exp(cmix__if_exp(pgm[func], pp),
                                  pstore, lstore, gstore),
                       rpp1, rpp2);
        pp = 0;
        break;

    case CALL_STMT:
        store = cmix_alloc_val(cmix__call_nspars(pgm[func],pp));
        for (n = 0; n < cmix__call_nspars(pgm[func], pp); n+=1)
          assign(store[n],
                 eval_exp(cmix__call_spar(pgm[func], pp, n),
                          pstore, lstore, gstore));
        assign(res,
               spec_exec_fun(cmix__call_fun(pgm[func], pp),
                             store));
        rpp = cmix_dealloc_store(store);
        switch (cmix__call_loc(pgm[func], pp))
          {
          case PARAMETER_VAR:
              lval = &pstore[cmix__call_var(pgm[func], pp)];
              break;
          case LOCAL_VAR:
              lval = &lstore[cmix__call_var(pgm[func], pp)];
              break;
          case GLOBAL_VAR:
              lval = &gstore[cmix__call_var(pgm[func], pp)];
          }
        assign(lval[0], res);
        pp += 1;
        break;
```

```
case _CALL_STMT:
    /*
     * Dynamic call:
     * Allocate store for static arguments and evaluate
     * these.  Reduce dynamic arguments.
     * Specialize called function with respec to
     * static arguments and static globals.
     * Update the store according to the collected
     * end-configuration stores.
     */
    store = cmix_alloc_val(cmix__call_nspars(pgm[func],pp));
    for (n = 0; n < cmix__call_nspars(pgm[func], pp); n+=1)
      assign(store[n],
             eval_exp(cmix__call_spar(pgm[func], pp, n),
                      pstore, lstore, gstore));
    rfunc
      = spec_pend_seen_call(cmix__call_fun(pgm[func], pp),
                            store, gstore);
    if (!rfunc)
      rfunc = spec_func(cmix__call_fun(pgm[func], pp),
                        store);
    rpp =
      spec_make_call(rfunc,
                     cmix__call_var(pgm[func], pp),
                     cmix__call_loc(pgm[func], pp),
                     n,
                     reduce_exps(cmix__call_dpars
                                 (pgm[func], pp),
                                 pstore, lstore, gstore));
    rpp = spec_pend_call(pp, rfunc, pstore, lstore, gstore);
    pp = 0;
    break;

case _RCALL_STMT:
    /*
     * Dynamic recursive call:
     * Allocate store for static arguments and evaluate
     * these.  Reduce dynamic arguments.
     * Specialize called function with respec to
     * static arguments and static globals.
     */
    store = cmix_alloc_val(cmix__call_nspars(pgm[func],pp));
    for (n = 0; n < cmix__call_nspars(pgm[func], pp); n+=1)
      assign(store[n],
```

```
                        eval_exp(cmix__call_spar(pgm[func], pp, n),
                                 pstore, lstore, gstore));
                    rfunc
                      = spec_pend_seen_call(cmix__call_fun(pgm[func], pp),
                                            store, gstore);
                    if (!rfunc)
                      rfunc = spec_func(cmix__call_fun(pgm[func], pp),
                                        store);
                    rpp =
                      spec_make_call(rfunc,
                                     cmix__call_var(pgm[func], pp),
                                     cmix__call_loc(pgm[func], pp),
                                     n,
                                     reduce_exps
                                     (cmix__call_dpars(pgm[func], pp),
                                      pstore, lstore, gstore));
                    pp += 1;
                    break;

                case _RETURN_STMT:
                    /*
                     * Dynamic return:
                     * Reduce the expression and build residual statement
                     */
                    rpp = spec_make_endconf(pstore, gstore);
                    rpp = spec_make_return(reduce_exp
                                           (cmix__return_exp(pgm[func], pp),
                                            pstore, lstore, gstore));
                    pp = 0;
                    break;

                default:
                    spec_error("unknown kind of syntax\n");
                }
            }
      }
    rpp = spec_close_pend();
    return spec_close_code();
#ifdef DEBUG_SEPC
    fprintf(stderr, "DEBUG_SPEC: End of specializer %d\n", func);
#endif
}
```

```
/*****************************************************************************
 * Function execution                                                        *
 *****************************************************************************/

Val
spec_exec_fun(int func, Val *pstore)
{
    int
      pp,                               /* Pending index                   */
      n,
      foo;

    Val
      *lstore,                          /* Local store                     */
      *store,                           /* Parameter store for calls       */
      res,                              /* Result temporary                */
      *lval;                            /* Left value pointer              */


    /* Initialization:
     * Allocate store for local variables,
     */
    lstore = cmix_alloc_store(cmix_local_decl(pgm[func]));
                                        /* Allocate local store    */
    pp = 1;

    /* Standard execution                                                    */
    while (pp)
      {
#ifdef DEBUG_SPEC
          fprintf(stderr, "Execution of %d in %d\n", pp,func);
#endif
          switch (cmix__stmt_kind(pgm[func], pp))
            {
            case ASSIGN_STMT:
                /*
                 * Assignments:
                 * Evaluate the expressions and perform an assignment
                 */
                switch (cmix__assign_loc(pgm[func], pp))
                  {
                  case PARAMETER_VAR:
                      lval = &pstore[cmix__assign_var(pgm[func], pp)];
                      break;
```

```
        case LOCAL_VAR:
            lval = &lstore[cmix__assign_var(pgm[func], pp)];
            break;
        case GLOBAL_VAR:
            lval = &gstore[cmix__assign_var(pgm[func], pp)];
            break;
        }
      for (n = 0; n < cmix__assign_nsindex(pgm[func], pp);
           n += 1)
        lval = &(val_to_ptr(lval[0])
                 [val_to_int
                  (eval_exp
                   (cmix__assign_index(pgm[func],pp, n),
                    pstore, lstore, gstore))]);
      assign(lval[0],eval_exp(cmix__assign_exp(pgm[func], pp),
                              pstore, lstore, gstore));
      pp += 1;
      break;

  case GOTO_STMT:
      /*
       * Goto:
       * Perform transition compression: jump to the target
       */
      pp = cmix__goto_label(pgm[func], pp);
      break;

  case IF_STMT:
      /*
       * If:
       * Evaluate the test-expression and perform
       * transition compression
       */
      goto SIF; /* Force next if to be specialization point      */
  SIF: if (val_to_int(eval_exp(cmix__if_exp(pgm[func], pp),
                               pstore, lstore, gstore)))
        pp = cmix__if_then(pgm[func], pp);
      else
        pp = cmix__if_else(pgm[func], pp);
      break;

  case CALL_STMT:
      /*
       * Call statement:
```

```
                 * Allocate store for arguments and evaluate these.
                 * Execute the function and update the local store
                 * with the result
                 */
                store = cmix_alloc_val(cmix__call_nspars(pgm[func],pp));
                for (n = 0; n < cmix__call_nspars(pgm[func], pp); n += 1)
                  assign(store[n],
                         eval_exp(cmix__call_spar(pgm[func], pp, n),
                                  pstore, lstore, gstore));
                assign(res,
                       spec_exec_fun(cmix__call_fun(pgm[func], pp), store));
                foo = cmix_dealloc_store(store);
                switch (cmix__call_loc(pgm[func], pp))
                  {
                  case PARAMETER_VAR:
                      lval = &pstore[cmix__call_var(pgm[func], pp)];
                      break;
                  case LOCAL_VAR:
                      lval = &lstore[cmix__call_var(pgm[func], pp)];
                      break;
                  case GLOBAL_VAR:
                      lval = &gstore[cmix__call_var(pgm[func], pp)];
                      break;
                  }
                assign(lval[0], res);
                pp += 1;
                break;

            case RETURN_STMT:
                /*
                 * Return:
                 * Reduce the expression and built residual statement
                 */
                return eval_exp(cmix__return_exp(pgm[func], pp),
                                pstore, lstore, gstore);
                break;

            default:
                spec_error("unknown kind of syntax\n");

            }
        }
    return int_to_val(0);
}
```

Appendix C

Annotated Core C Program Specializer

```
Fun * pgm;
_Val * gstore;

_Fun * specialize (Fun * program; _Val * parameter; )
{
        _int  foo;

        1: assign pgm = program;
        2: _assign gstore =
          _ecall cmix_alloc_store (@ecall cmix_global_decl (index pgm[0]));
        3: _assign foo = _ecall spec_init_global
                                (@ecall cmix_global_decl(index pgm [0]));
        4: _rcall foo = spec_func (1, parameter);
        5: _return _ecall spec_get_pgm ();
}

_int  spec_func (int  func; _Val * pstore; )
{
    int  pp;
    _Val * lstore;
    _int  pp0;
    int  n;
    _int  rfunc;
    _int  rpp;
    _int  rpp1;
    _int  rpp2;
    _Val * store;
    _Val  res;
    _Val * lval;
```

```
int  genvar_0;
int  genvar_1;
int  genvar_2;

1: _assign lstore =
  _ecall cmix_alloc_store (@ecall cmix_local_decl (index pgm[func]));
2: _assign rpp = _ecall spec_init_code
                        (@func,
                         @ecall cmix_fun_name(index pgm[func]),
                         @ecall cmix_fun_type(index pgm[func]),
                         @ecall cmix_param_decl(index pgm[func]),
                         @ecall cmix_local_decl(index pgm[func]),
                         pstore, gstore);
3: _assign rpp = _ecall spec_init_pending (pstore, lstore, gstore);
4: _goto 5;
5: _if (_ecall spec_pending ()) 6 122;
6: _assign pp0 = _ecall spec_pend_pp ();
7: _assign pstore = _ecall spec_pend_pstore ();
8: _assign lstore = _ecall spec_pend_lstore ();
9: _assign gstore = _ecall spec_pend_gstore ();
10: _assign rpp = _ecall spec_pend_processed ();
11: _assign rpp = _ecall spec_make_label (rpp);
12: assign pp = 1;
13: if (bop pp <= ecall cmix_num_stmt (index pgm[func])) 14 20;
14: _if (_bop @pp == pp0) 21 15;
15: assign pp = bop pp + 1;
16: if (uop ! ecall spec_point (pp, index pgm[func])) 17 19;
17: assign pp = bop pp + 1;
18: goto 16;
19: goto 13;
20: _return _ecall spec_error(@"Program point not found");
21: if (pp) 22 121;
22: assign genvar_2 = ecall cmix__stmt_kind (index pgm[func], pp);
23: if (bop genvar_2 == 1) 33 24;
24: if (bop genvar_2 == 2) 51 25;
25: if (bop genvar_2 == 3) 54 26;
26: if (bop genvar_2 == 4) 56 27;
27: if (bop genvar_2 == 7) 60 28;
28: if (bop genvar_2 == 8) 66 29;
29: if (bop genvar_2 == 9) 71 30;
30: if (bop genvar_2 == 10) 91 31;
31: if (bop genvar_2 == 11) 104 32;
32: if (bop genvar_2 == 6) 116 120;
33: assign genvar_0 = ecall cmix__assign_loc (index pgm[func], pp);
```

```
34: if (bop genvar_0 == 1) 37 35;
35: if (bop genvar_0 == 2) 39 36;
36: if (bop genvar_0 == 3) 41 43;
37: _assign lval =
     _addr & _index pstore[@ecall cmix__assign_var (index pgm[func], pp)];
38: goto 43;
39: _assign lval =
    _addr & _index lstore[@ecall cmix__assign_var (index pgm[func], pp)];
40: goto 43;
41: _assign lval =
    _addr & _index gstore[@ecall cmix__assign_var (index pgm[func], pp)];
42: goto 43;
43: assign n = 0;
44: if (bop n < ecall cmix__assign_nsindex (index pgm[func], pp)) 45 48;
45: _assign lval =
    _addr & _index _ecall val_to_ptr (_index lval[@0])
                   [_ecall val_to_int
                    (_ecall eval_exp
                     (@ecall cmix__assign_index (index pgm[func], pp,n),
                      pstore, lstore, gstore))];
46: assign n = bop n + 1;
47: goto 44;
48: _assign _index lval[@0] =
    _ecall eval_exp (@ecall cmix__assign_exp (index pgm[func], pp),
                     pstore, lstore, gstore);
49: assign pp = bop pp + 1;
50: goto 120;
51: _assign rpp =
    _ecall spec_make_assign
      (_ecall reduce_exp (@ecall cmix__assign_lexp (index pgm[func], pp),
                          pstore, lstore, gstore),
       _ecall reduce_exp (@ecall cmix__assign_exp (index pgm[func], pp),
                          pstore, lstore, gstore));
52: assign pp = bop pp + 1;
53: goto 120;
54: assign pp = ecall cmix__goto_label (index pgm[func], pp);
55: goto 120;
56: _assign rpp =
    _ecall spec_pend_insert
           (@ecall cmix__goto_label (index pgm[func], pp),
            @ecall cmix__goto_pdead (index pgm[func], pp),
            @ecall cmix__goto_ldead (index pgm[func], pp),
            @ecall cmix__goto_gdead (index pgm[func], pp),
            pstore, lstore, gstore);
```

```
57: _assign rpp = _ecall spec_make_goto (rpp);
58: assign pp = 0;
59: goto 120;
60: _goto 61 {0L 30L 0L};
61: _if (_ecall val_to_int
            (_ecall eval_exp (@ecall cmix__if_exp (index pgm[func], pp),
                              pstore, lstore, gstore)))
    62 { 0L 30L 0L } 64 { 0L 30L 0L };
62: assign pp = ecall cmix__if_then (index pgm[func], pp);
63: goto 65;
64: assign pp = ecall cmix__if_else (index pgm[func], pp);
65: goto 120;
66: _assign rpp2 =
    _ecall spec_pend_insert (@ecall cmix__if_else (index pgm[func], pp),
                             @ecall cmix__ife_pdead (index pgm[func], pp),
                             @ecall cmix__ife_ldead (index pgm[func], pp),
                             @ecall cmix__ife_gdead (index pgm[func], pp),
                             pstore, lstore, gstore);
67: _assign rpp1 =
    _ecall spec_pend_insert (@ecall cmix__if_then (index pgm[func], pp),
                             @ecall cmix__ift_pdead (index pgm[func], pp),
                             @ecall cmix__ift_ldead (index pgm[func], pp),
                             @ecall cmix__ift_gdead (index pgm[func], pp),
                             pstore, lstore, gstore);
68: _assign rpp =
    _ecall spec_make_if (_ecall reduce_exp
                            (@ecall cmix__if_exp (index pgm[func], pp),
                             pstore, lstore, gstore),
                         rpp1, rpp2);
69: assign pp = 0;
70: goto 120;
71: _assign store =
    _ecall cmix_alloc_val (@ecall cmix__call_nspars (index pgm[func],pp));
72: assign n = 0;
73: if (bop n < ecall cmix__call_nspars (index pgm[func], pp)) 74 77;
74: _assign _index store[@n] =
    _ecall eval_exp (@ecall cmix__call_spar (index pgm[func], pp, n),
                     pstore, lstore, gstore);
75: assign n = bop n + 1;
76: goto 73;
77: _rcall res =
    spec_exec_fun (ecall cmix__call_fun (index pgm[func], pp), store);
78: _assign rpp = _ecall cmix_dealloc_store (store);
79: assign genvar_1 = ecall cmix__call_loc (index pgm[func], pp);
```

```
 80: if (bop genvar_1 == 1) 83 81;
 81: if (bop genvar_1 == 2) 85 82;
 82: if (bop genvar_1 == 3) 87 88;
 83: _assign lval =
     _addr & _index pstore[@ecall cmix__call_var (index pgm[func], pp)];
 84: goto 88;
 85: _assign lval =
     _addr & _index lstore[@ecall cmix__call_var (index pgm[func], pp)];
 86: goto 88;
 87: _assign lval =
     _addr & index gstore[@ecall cmix__call_var (index pgm[func], pp)];
 88: _assign _index lval[@0] = res;
 89: assign pp = bop pp + 1;
 90: goto 120;
 91: _assign store =
     _ecall cmix_alloc_val (@ecall cmix__call_nspars(index pgm[func], pp));
 92: assign n = 0;
 93: if (bop n < ecall cmix__call_nspars (index pgm[func], pp)) 94 97;
 94: _assign _index store[@n] =
     _ecall eval_exp (@ecall cmix__call_spar (index pgm[func], pp, n),
                      pstore, lstore, gstore);
 95: assign n = bop n + 1;
 96: goto 93;
 97: _assign rfunc =
     _ecall spec_pend_seen_call (@ecall cmix__call_fun(index pgm[func],pp),
                                 store, gstore);
 98: _if (_uop ! rfunc) 99 100;
 99: _rcall rfunc =
     spec_func (ecall cmix__call_fun (index pgm[func], pp), store);
100: _assign rpp =
     _ecall spec_make_call(rfunc,
                           @ecall cmix__call_var (index pgm[func], pp),
                           @ecall cmix__call_loc (index pgm[func], pp),
                           @n,
                           _ecall reduce_exps (@ecall cmix__call_dpars
                                               (index pgm[func], pp),
                                               pstore, lstore, gstore));
101: _assign rpp =
      _ecall spec_pend_call (@pp, rfunc, pstore, lstore, gstore);
102: assign pp = 0;
103: goto 120;
104: _assign store =
      _ecall cmix_alloc_val (@ecall cmix__call_nspars(index pgm[func], pp));
105: assign n = 0;
```

```
    106: if (bop n < ecall cmix__call_nspars (index pgm[func], pp)) 107 110;
    107: _assign _index store[@n] =
         _ecall eval_exp (@ecall cmix__call_spar (index pgm[func], pp, n),
                          pstore, lstore, gstore);
    108: assign n = bop n + 1;
    109: goto 106;
    110: _assign rfunc =
         _ecall spec_pend_seen_call(@ecall cmix__call_fun(index pgm[func],pp),
                                    store, gstore);
    111: _if (_uop ! rfunc) 112 113;
    112: _rcall rfunc =
         spec_func (ecall cmix__call_fun (index pgm[func], pp), store);
    113: _assign rpp =
         _ecall spec_make_call (rfunc,
                                @ecall cmix__call_var (index pgm[func], pp),
                                @ecall cmix__call_loc (index pgm[func], pp),
                                @n,
                                _ecall reduce_exps
                                (@ecall cmix__call_dpars (index pgm[func],pp),
                                 pstore, lstore, gstore));
    114: assign pp = bop pp + 1;
    115: goto 120;
    116: _assign rpp = _ecall spec_make_endconf (pstore, gstore);
    117: _assign rpp =
         _ecall spec_make_return
           (_ecall reduce_exp (@ecall cmix__return_exp (index pgm[func], pp),
                               pstore, lstore, gstore));
    118: assign pp = 0;
    119: goto 120;
    120: goto 21;
    121: _goto 5 { 0L 31L 0L};
    122: _assign rpp = _ecall spec_close_pend();
    123: _return _ecall spec_close_code ();
}

_Val  spec_exec_fun (int  func; _Val * pstore; )
{
    int  pp;
    int  n;
    _Val * lstore;
    _Val * store;
    _Val  res;
    _Val * lval;
    int  genvar_3;
```

```
int  genvar_4;
int  genvar_5;
_int foo;

1: _assign lstore =
   _ecall cmix_alloc_store (@ecall cmix_local_decl (index pgm[func]));
2: assign pp = 1;
3: if (pp) 4 60;
4: assign genvar_5 = ecall cmix__stmt_kind (index pgm[func], pp);
5: if (bop genvar_5 == 1) 10 6;
6: if (bop genvar_5 == 3) 28 7;
7: if (bop genvar_5 == 7) 30 8;
8: if (bop genvar_5 == 9) 36 9;
9: if (bop genvar_5 == 5) 57 59;
10: assign genvar_3 = ecall cmix__assign_loc (index pgm[func], pp);
11: if (bop genvar_3 == 1) 14 12;
12: if (bop genvar_3 == 2) 16 13;
13: if (bop genvar_3 == 3) 18 20;
14: _assign lval =
    _addr & _index pstore[@ecall cmix__assign_var (index pgm[func], pp)];
15: goto 20;
16: _assign lval =
    _addr & _index lstore[@ecall cmix__assign_var (index pgm[func], pp)];
17: goto 20;
18: _assign lval =
    _addr & _index gstore[@ecall cmix__assign_var (index pgm[func], pp)];
19: goto 20;
20: assign n = 0;
21: if (bop n < ecall cmix__assign_nsindex (index pgm[func], pp)) 22 25;
22: _assign lval =
     _addr & _index _ecall val_to_ptr
               (index lval[@0])
                 [_ecall val_to_int
                  (_ecall eval_exp
                   (@ecall cmix__assign_index (index pgm[func], pp, n),
                    pstore, lstore, gstore))];
23: assign n = bop n + 1;
24: goto 21;
25: _assign _index lval[@0] =
    _ecall eval_exp (@ecall cmix__assign_exp (index pgm[func], pp),
                     pstore, lstore, gstore);
26: assign pp = bop pp + 1;
27: goto 59;
28: assign pp = ecall cmix__goto_label (index pgm[func], pp);
```

```
    29: goto 59;
    30: _goto 31 {0L 30L 0L};
    31: _if (_ecall val_to_int (_ecall eval_exp
                               (@ecall cmix__if_exp (index pgm[func], pp),
                                pstore, lstore, gstore)))
        32 {0L 30L 0L } 34  {0L 30L 0L};
    32: assign pp = ecall cmix__if_then (index pgm[func], pp);
    33: goto 35;
    34: assign pp = ecall cmix__if_else (index pgm[func], pp);
    35: goto 59;
    36: _assign store =
        _ecall cmix_alloc_val (@ecall cmix__call_nspars(index pgm[func], pp));
    37: assign n = 0;
    38: if (bop n < ecall cmix__call_nspars (index pgm[func], pp)) 39 42;
    39: _assign _index store[@n] =
        _ecall eval_exp (@ecall cmix__call_spar (index pgm[func], pp, n),
                         pstore, lstore, gstore);
    40: assign n = bop n + 1;
    41: goto 38;
    42: _rcall res =
        spec_exec_fun (@ecall cmix__call_fun (index pgm[func], pp), store);
    43: _assign foo = _ecall cmix_dealloc_store (store);
    44: assign genvar_4 = ecall cmix__call_loc (index pgm[func], pp);
    45: if (bop genvar_4 == 1) 48 46;
    46: if (bop genvar_4 == 2) 50 47;
    47: if (bop genvar_4 == 3) 52 54;
    48: _assign lval =
        _addr & _index pstore[@ecall cmix__call_var (index pgm[func], pp)];
    49: goto 54;
    50: _assign lval =
        _addr & _index lstore[@ecall cmix__call_var (index pgm[func], pp)];
    51: goto 54;
    52: _assign lval =
        _addr & _index gstore[@ecall cmix__call_var (index pgm[func], pp)];
    53: goto 54;
    54: _assign _index lval[@0] = res;
    55: assign pp = bop pp + 1;
    56: goto 59;
    57: _return _ecall eval_exp (@ecall cmix__return_exp(index pgm[func], pp),
                                 pstore, lstore, gstore);
    58: goto 59;
    59: goto 3;
    60: _return _ecall int_to_val (@0);
}
```

Appendix D

The Power Compiler

```
/* C-MIX produced */
#include <stdio.h>
#include "asyntax.h"
#include "value.h"
#include "expr.h"
#include "memory.h"
#include "pend.h"
#include "codegen.h"
#include "cmix.h"
#include "spec.h"


Fun    *specialize_1_0 (Val *);
int     spec_func_2_1 (Val *);

int     cmix_endconf;
Val    *gstore;


Fun    *
specialize_1_0 (Val * parameter)
{
   int     foo;

 lab1:
   gstore = cmix_alloc_store (cmix_make_dec (VOID_DECL));
   foo = spec_init_global (cmix_make_dec (VOID_DECL));
   foo = spec_func_2_1 (parameter);
   cmix_endconf = 3;
   return spec_get_pgm ();
```

```
}
int
spec_func_2_1 (Val * pstore)
{
   Val    *lstore;
   int     pp0;
   int     rfunc;
   int     rpp;
   int     rpp1;
   int     rpp2;
   Val    *store;
   Val     res;
   Val    *lval;

 lab1:
   lstore = cmix_alloc_store
     (cmix_make_decl (VAR_DECL, "pow", 1, 0, cmix_make_dec (TYPE_INT),
      cmix_make_dec (VOID_DECL)));
   rpp = spec_init_code (1, "power", cmix_make_dec (TYPE_INT),
           cmix_make_decl (VAR_DECL, "n", 0, 0, cmix_make_dec (TYPE_INT),
           cmix_make_decl (VAR_DECL, "x", 1, 0, cmix_make_dec (TYPE_INT),
           cmix_make_dec (VOID_DECL))),
           cmix_make_decl (VAR_DECL, "pow", 1, 0, cmix_make_dec (TYPE_INT),
           cmix_make_dec (VOID_DECL)), pstore, gstore);
   rpp = spec_init_pending (pstore, lstore, gstore);
   goto lab5;
 lab5:
   if (spec_pending ())
   {
    lab6:
      pp0 = spec_pend_pp ();
      pstore = spec_pend_pstore ();
      lstore = spec_pend_lstore ();
      gstore = spec_pend_gstore ();
      rpp = spec_pend_processed ();
      rpp = spec_make_label (rpp);
      if (1 == pp0)
      {
       lab13:
         rpp = spec_make_assign
           (cmix_make_id_exp (BT_STATIC, 0, 2),
            cmix__make_lift_exp (cmix_make_ecall (BT_STATIC, "int_to_val",
                                  cmix_make_exps (cmix_make_int_exp (1),
                                                  cmix_make_null_exp ()))));
```

```
        goto lab15;
      lab15:
        if (val_to_int (pstore[0]))
        {
         lab16:
          rpp = spec_make_assign
            (cmix_make_id_exp (BT_STATIC, 0, 2),
             cmix_make_bop (BT_STATIC, 16, cmix_make_id_exp (BT_STATIC, 0, 2),
                            cmix_make_id_exp (BT_STATIC, 0, 1)));
          lval = &pstore[0];
          lval[0] = int_to_val (val_to_int (pstore[0]) -
                                val_to_int (int_to_val (1)));
          goto lab15;
        }
        else
        {
         lab20:
          rpp = spec_make_endconf (pstore, gstore);
          rpp = spec_make_return (cmix_make_id_exp (BT_STATIC, 0, 2));
          goto lab5;
        }
      }
      else
      {
       lab23:
        cmix_endconf = 1;
        return spec_error ("Program point not found");
      }
    }
    else
    {
     lab25:
      rpp = spec_close_pend ();
      cmix_endconf = 2;
      return spec_close_code ();
    }

}
```

Appendix E

The Poolish Form Interpreter

```
/*
 *
 * File:   int.c
 *      Author:   Lars Ole Andersen (lars@diku.dk)
 *      Created:  Wed Dec 18 22:09:52 1991
 *      Modified: Wed Dec 18 22:09:52 1991
 *
 *      Content:  Poolish form Interpreter
 *
 */

#ifndef _CMIX
#include <stdio.h>
#endif
#include "intr.h"

extern int
  readnum(void), printnum(int);
int
  interpret(int *);

int
  s[100];                               /* The stack                    */
int
  stg[100];                             /* Variable store               */


int
interpret(int *p)
{
    int pp;                             /* Program counter              */
```

```
int sp;                             /* Stack counter                    */
int foo;

/* Initialize                                                           */
pp = 1;
sp = 0;

/* Interpretation loop                                                  */
while (p[pp] != HALT)
  switch (p[pp])
    {
    case CON:
        sp += 1; s[sp] = p[pp+1]; pp += 2; break;
    case LVAL:
        sp += 1; s[sp] = p[pp+1]; pp += 2; break;
    case RVAL:
        sp += 1; s[sp] = stg[p[pp+1]]; pp += 2;     break;
    case ADD:
        s[sp-1] = s[sp-1] + s[sp]; sp -= 1; pp += 1; break;
    case SUB:
        s[sp-1] = s[sp-1] - s[sp]; sp -= 1; pp += 1; break;
    case MUL:
        s[sp-1] = s[sp-1] * s[sp]; sp -= 1; pp += 1; break;
    case DVD:
        s[sp-1] = s[sp-1] / s[sp]; sp -= 1; pp += 1; break;
    case J:
        pp = p[pp+1]; break;
    case JN:
        sp -= 1;
        if (s[sp-1] < 0)
          pp = p[pp+1];
        else
          pp += 2;
        break;
    case JP:
        sp -= 1;
        if (s[sp+1] > 0)
          pp = p[pp+1];
        else
          pp += 2;
        break;
    case JZ:
        sp -= 1;
        if (s[sp+1] == 0)
```

```
            pp = p[pp+1];
        else
            pp += 2;
        break;
    case JNZ:
        sp -= 1;
        if (s[sp+1] <= 0)
            pp = p[pp+1];
        else
            pp += 2;
        break;
    case JPZ:
        sp -= 1;
        if (s[sp+1] >= 0)
            pp = p[pp+1];
        else
            pp += 2;
        break;
    case JNP:
        sp -= 1;
        if (s[sp+1] != 0)
            pp = p[pp+1];
        else
            pp += 2;
        break;
    case GET:
        stg[s[sp]] = readnum();
        sp -= 1;
        pp += 1;
        break;
    case PUT:
        foo = printnum(s[sp]);
        sp -= 1;
        pp += 1;
        break;
    case ASGN:
        stg[s[sp-1]] = s[sp]; sp -= 2; pp += 1;     break;
    default:
        pp = HALT;
    }
  return 0;
}
```

Appendix F

The Poolish Primes in C

```
/* C-MIX produced */
#include <stdio.h>
#include "asyntax.h"
#include "value.h"
#include "expr.h"
#include "memory.h"
#include "pend.h"
#include "codegen.h"
#include "cmix.h"
#include "spec.h"


int     interpret_1_0 (void);

int     cmix_endconf;
int     s_0;
int     s_1;
int     s_2;
int     s_3;
/* 97 more ... */
int     stg[100];


int
interpret_1_0 (void)
{
   int     foo;

 lab1:
   s_1 = 1;
   stg[s_1] = readnum ();
```

```
  s_1 = 2;
  s_2 = 2;
  stg[s_1] = s_2;
  s_1 = 2;
  foo = printnum (s_1);
  s_1 = 3;
  foo = printnum (s_1);
  s_1 = 3;
  s_2 = 3;
  stg[s_1] = s_2;
  s_1 = stg[2];
  s_2 = stg[1];
  s_1 = s_1 - s_2;
  if (s_1 >= 0)
     goto lab17;
  else
     goto lab19;
lab17:
  cmix_endconf = 1;
  return 0;
lab19:
  s_1 = 3;
  s_2 = stg[3];
  s_3 = 2;
  s_2 = s_2 + s_3;
  stg[s_1] = s_2;
  s_1 = 4;
  s_2 = 0;
  stg[s_1] = s_2;
  s_1 = 5;
  s_2 = 1;
  stg[s_1] = s_2;
  s_1 = 5;
  s_2 = stg[5];
  s_3 = 2;
  s_2 = s_2 + s_3;
  stg[s_1] = s_2;
  s_1 = stg[5];
  s_2 = stg[3];
  s_1 = s_1 - s_2;
  if (s_1 >= 0)
     goto lab39;
  else
     goto lab72;
```

```
lab39:
  s_1 = stg[4];
  if (s_1 <= 0)
    {
     lab41:
       s_1 = stg[3];
       foo = printnum (s_1);
       s_1 = 2;
       s_2 = stg[2];
       s_3 = 1;
       s_2 = s_2 + s_3;
       stg[s_1] = s_2;
       s_1 = stg[2];
       s_2 = stg[1];
       s_1 = s_1 - s_2;
       if (s_1 >= 0)
          goto lab17;
       else
          goto lab19;
    }
  else
    {
     lab52:
       s_1 = 3;
       s_2 = stg[3];
       s_3 = 2;
       s_2 = s_2 + s_3;
       stg[s_1] = s_2;
       s_1 = 4;
       s_2 = 0;
       stg[s_1] = s_2;
       s_1 = 5;
       s_2 = 1;
       stg[s_1] = s_2;
       s_1 = 5;
       s_2 = stg[5];
       s_3 = 2;
       s_2 = s_2 + s_3;
       stg[s_1] = s_2;
       s_1 = stg[5];
       s_2 = stg[3];
       s_1 = s_1 - s_2;
       if (s_1 >= 0)
          goto lab39;
```

```
  else
     goto lab72;
lab72:
  s_1 = stg[3];
  s_2 = stg[5];
  s_1 = s_1 / s_2;
  s_2 = stg[5];
  s_1 = s_1 * s_2;
  s_2 = stg[3];
  s_1 = s_1 - s_2;
  if (s_1 != 0)
    {
     lab80:
       s_1 = 5;
       s_2 = stg[5];
       s_3 = 2;
       s_2 = s_2 + s_3;
       stg[s_1] = s_2;
       s_1 = stg[5];
       s_2 = stg[3];
       s_1 = s_1 - s_2;
       if (s_1 >= 0)
          goto lab39;
       else
          goto lab72;
    }
  else
    {
     lab89:
       s_1 = 4;
       s_2 = stg[4];
       s_3 = 1;
       s_2 = s_2 + s_3;
       stg[s_1] = s_2;
       s_1 = 5;
       s_2 = stg[5];
       s_3 = 2;
       s_2 = s_2 + s_3;
       stg[s_1] = s_2;
       s_1 = stg[5];
       s_2 = stg[3];
       s_1 = s_1 - s_2;
       if (s_1 >= 0)
          goto lab39;
```

```
            else
               goto lab72;
         }
      }

}
```