A Framework for On-line Partial Evaluation

by

Gordon J. Vreugdenhil

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 1996

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

The term *partial evaluation* describes a class of program transformation techniques. The heart of these techniques is to transform programs by incorporating portions of known run-time data into the program. The resulting program has been "partially" evaluated — some of the actions of the program can be performed at compile-time due to the known data. There are two general classes of known data that can be used by such a process. The first class is composed of data that is implicit in the production of the program; examples include textual constants, macro expanded values, type tag values, method dispatch tables, etc. Some amount of such data occurs frequently in high-level programs. The second category is composed of data that is explicitly provided at compile time. Such data can be used to create customized versions of very general programs such as ray-tracing and numerical modeling systems.

In this thesis we propose a formal framework for an on-line partial evaluation system. The underlying model for values in the partial evaluator is not restricted to finite-height lattices; the termination of the evaluator depends on the convergence of operations, rather than on a restricted model for values in the system. The proposed framework clearly separates the partial evaluation algorithm from the abstract domains used for representing information during the evaluation, allowing a wide variety of evaluations to be effected by the same core algorithm. The partial evaluation algorithm that is proposed as part of the framework is a polyvariant on-line algorithm that makes effective use of the static information present in program source while preserving soundness and termination. The thesis presents careful proofs of termination and soundness based on characterizations of behaviour under the natural semantics. The key to the algorithm is recognizing when exact analysis is safe with regards to termination and when a more conservative approximation is needed.

The actual on-line algorithm depends only on the properties of the abstract domains, not on particular choices of abstraction. The abstract domains allow the partial evaluation algorithm to take advantage of safe computations whenever possible. The overall algorithm we propose compares favourably to other partial evaluation systems in its ability to capture information present in the program, and the ability of the system to execute without any human intervention other than an indication of how much the system is permitted to increase the size of resulting program. The ability of a general system to generate reasonable results without human intervention is a key advantage that is a prerequisite for having this type of technology applied in real systems.

Acknowledgements

As with any Ph.D. student, I have had the opportunity over the last number of years to interact with many outstanding people. It is only a small measure of my appreciation to acknowledge some of these people here.

First of all, I wish to acknowledge the invaluable assistance of my supervisor, Gord Cormack. Gord supervised both my Masters thesis and my Ph.D. work so we ended up working together for eight years. His insights about many aspects of computer science helped to keep my work in focus. Particularly in the last year of work on my Ph.D. thesis, Gord spent substantial amounts of time working through proofs and helping to develop and improve them.

I would like to thank Dave Mason and Glenn Paulley, fellow Ph.D. students, for the many great discussions over the last number of years. Dave and Glenn have been not only sounding boards, but good friends who made the time spent at Waterloo enjoyable. I also want to express my appreciation for the discussions with other members, past and present, of the Programming Languages Group at Waterloo. In particular, I would like to thank Charlie Clarke, Dennis Vadura, Dominic Duggan, and Peter Buhr.

I would like to give a special note of thanks to the other members of my defense committee, Don Cowan, Rudy Seviora, and my external examiner, Jim Cordy. Their insights and comments helped to improve the clarity and precision of the thesis.

There are many other members of the Computer Science Department at Waterloo that provided encouragement during the process. In particular, the support provided by Byron Weber Becker, Naomi Nishimura, Prabhakar Ragde, and Grant Weddell was deeply appreciated. Of course, I can't forget the often taken-for-granted support staff, especially Wendy Rush who helped me to stay sane at various times.

Finally, I want to acknowledge the support and love that my wife, Janet, provided throughout the years of study. Her patience and support encouraged me and made it possible to stay focussed on my work during times of frustration.

Contents

List of Figures

Chapter 1

Introduction

1.1 Goals and Directions

Automatic program transformations are important in the practice of modern computer science. Programmers generally take for granted that compilers and other program transformation systems are correct and that compilers perform "good" transformations. Although most program transformations occur during compilation, it is increasingly important to express program transformations that support various kinds automated reasoning. Such transformations range from program specification, to real-time system behaviour, to dealing with changes to legacy code. In addition to the more recent concerns, the more traditional roles of program optimization continue to be very important in areas such as molecular modeling, weather systems, fluid dynamics, full motion animation, etc.

Answering questions about program behaviour is a fundamental aspect of nearly all program transformation techniques. This thesis proposes a framework for program transformation that is based on performing source language to source language code transformations that exploit information present in the original source program. The tradeoff for this focus lies in an increase in computation at compile-time and a probable increase in the size of the resulting executable program. The focus of our work is in a framework for program analysis. The framework that we propose can be used as a tool for answering various questions about program behaviour; the collected information can in turn be used for various types of transformations.

We propose the use of partial evaluation and symbolic execution techniques to regularize and formalize questions about program behaviour. The proposed framework generalizes the analysis methodology adopted by most comparable systems. There are three main areas of contribution presented in this work. First, a formal foundation for partial evaluation is presented. The foundation determines how abstract values within the system can be modeled. Prior work has required that such models be formed from finite-height lattices in order to preserve termination for the evaluator. Our analytic approach is based on the analysis techniques of Cousot and Cousot [27] and preserves termination without restricting the underlying model to finite height lattices. This approach is a general application of interval analysis and has reasonable extensions to non-integer domains such as structured types.

The second area of contribution is a partial evaluation algorithm that uses the formal model and that differentiates between types of information within the system. The algorithm allows very accurate operations on values when there is no risk of divergence and applies more conservative operations when needed in order to guarantee convergence. We present proofs of termination and soundness for our algorithm and discuss the general time complexity of the framework.

The final area of contribution deals with abstract domains for modeling integer and structural information. The structural model was motivated by the work of Hendren [39] and Launchbury [54], while the abstract integer model is based on work by Cousot and Cousot [27].

The overall design of the system separates the language specific foundation of the interpreter and the methods for performing the analysis. Such a design allows one to change easily the types of analysis performed by the system without having to change the underlying interpretation system. The overall system compares favourably to other partial evaluation systems in its ability to capture information present in the program, and in the system's ability to execute without any human intervention other than an indication of how much the system is permitted to increase the size of resulting program. The ability of a general system to generate reasonable results without human intervention is a key advantage that is a prerequisite for having this type of technology applied in real systems.

The remainder of this chapter introduces the general concepts of compilers, optimization, and abstract interpretation while Chapter 2 introduces partial evaluation and applications of partial evaluation. Chapter 3 presents the framework that we

have developed. There are two important parts to this presentation: the requirements for the abstract models used by our algorithm, and the algorithm itself. Chapter 4 presents a formal analysis of the algorithm and includes proofs of termination, soundness, and correctness of transformed expressions, as well as a discussion of the complexity of algorithm. In Chapter 5 we develop particular abstract models for integer and structural domains and discuss other possible models. Chapter 6 deals with a number of issues related to implementing the framework and includes a discussion of the prototype system that we have developed. Other issues, not directly related to our implementation, are also discussed. These include problems with side-effects, methods for producing high quality residuals, and separate compilation.

1.2 Compilers and Interpreters

The basic difference between interpretation and compilation is that an interpreter executes programs by translating a single line of a program, performing the required action, and then going on to the next line. After each translation and action, the interpreter throws away the translation, so if the interpreter encounters the same line again later, the line must be translated again. A compiler takes the original program and translates the entire program into an executable form that may then be used without further translation.

The interpretation/compilation border in real systems is not that well defined of course. Inherently, every real program is interpreted – the actual processor interprets a sequence of bits as an instruction to perform a particular action, then interprets the next bit sequence, etc. It is important that one does not assume that all compilers produce code that requires no further interpretation and it is equally important not to assume that an interpreter never compiles code.

A compiler is simply a program that transforms data according to some set of rules. Data transformations are not "magic"; any program can be seen as a data transformer for at least a trivial data set. The reason that people become confused about compilers is that although the result of the compiler can be understood as data, the result is not passive but rather is itself a data transformer.

The classic compiler structure [3] is composed of a number of phases or layers as shown in Figure 1.2.1 The first phases are syntactic or *lexical* analysis and parsing. These two phases insure that the program is structurally correct with respect to

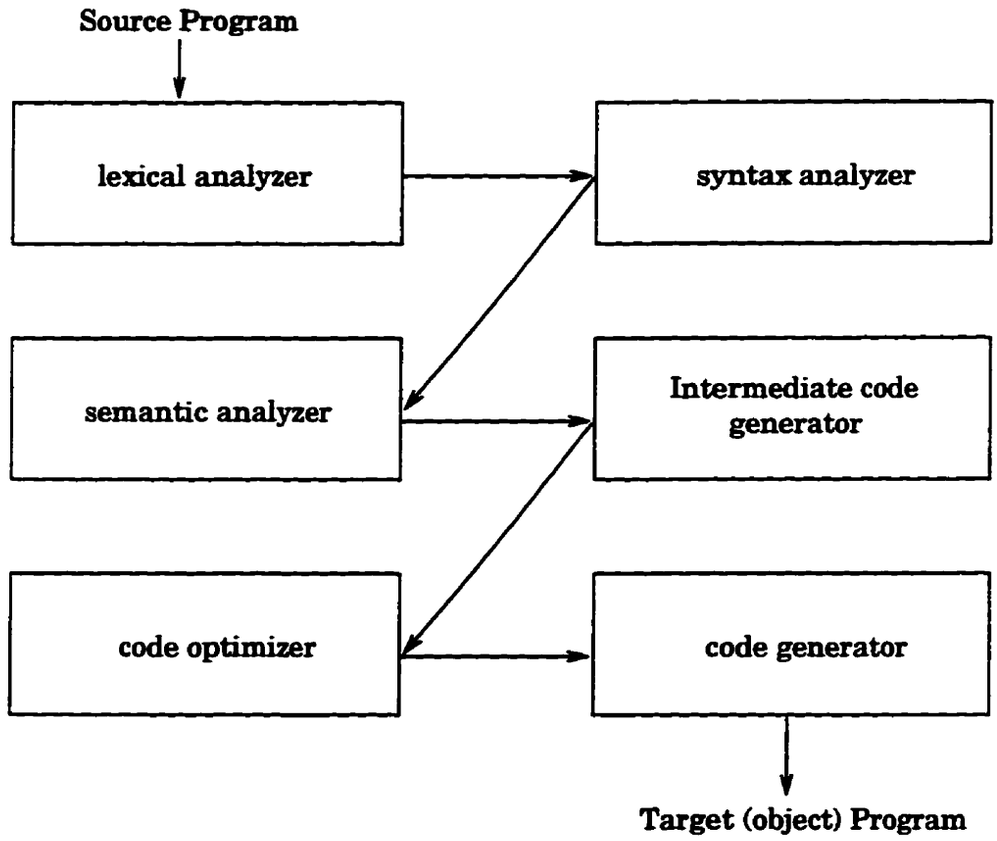


Figure 1.2.1: A typical compiler

the language definition (assuming, of course, that the compiler correctly implements the language specification). The result of these two phases is an intermediate form usually represented as a parse tree or some intermediate language. The semantic analysis normally checks that language constraints are satisfied. Such constraints may include type safety, assignment rules, etc. After (or more typically, during) semantic analysis, an intermediate form of the program is produced. Intermediate forms generally remove source language syntactic (and possibly some of the semantic) constraints and are constructed to be amenable to manipulation for the remaining phases. The optimizer performs transformations on the intermediate form and produces a semantically equivalent intermediate representation that is "better" according to some set of criteria. The name "optimizer" is somewhat misleading – it is extremely rare that an optimized program is in fact *optimal* in any formal sense. "Optimizations" are in reality "code-improving transformations", but we will retain the common terminology for the sake of clarity. The final step after optimization is the generation of the target code.

The research presented in this thesis is directed primarily at the optimization phase of the compiler although the approach could be used for code generation and semantic analysis as well.

1.3 Optimization and Interpretation

The optimization methods presented in this thesis derive information about the source code and makes use of this derived information when performing code transformations. Methods for deriving information about programs rely on some sort of interpretation of the source code. This interpretation cannot normally be a full execution of the program since we generally do not know run-time arguments to the program when we are compiling the program. Compile-time interpretation can only approximate the run-time behaviour of the program if there is any information that is not present at compile-time.

Simple examples of such interpretations are the common optimizations of constant folding and constant propagation. If a compiler encounters an expression such as `(a + b + 2 + 4)` within a program, it is generally safe to transform the calculation by folding the two constants into a single constant, resulting in the expression `(a + b + 6)`. It is important to note that such transformations are not *always* safe.

For example, on a machine with 8-bit two's complement arithmetic, folding `(a + b + 120 + 20)` to `(a + b + 140)` would not be safe since the constant `140` is not representable in 8-bit two's complement notation. The run-time semantics of the expression may be correct however since the programmer may have a priori knowledge that the result of `(a + b)` will always be below `-12`. Even this a priori knowledge however, relies on the assumption that expressions are evaluated in left to right order.

Constant propagation is a similar technique but is performed across expressions. If at a certain point in an imperative program, a variable is assigned a constant, we can replace uses of that variable in the following code with the constant value until the point in the program at which the variable is assigned some other value. Note that the code "following" an assignment depends on the *run-time* behaviour of the program – for example, in general *all* code within a loop "follows" every statement in the loop. For example, within the loop:

```
x := 5;
for i := 1 to 10 do
   y := x;
   output(y);
   x := x + 1;
od;
```

it would not be correct to remove the assignment of `x` to `y` and replace the `output(y)` with `output(5)` since the assignment statement `y := x` follows not only the statement `x := 5`, but also follows the statement `x := x + 1` which occurs textually at the end of the loop.

Complicating matters in constant propagation analysis is the fact that there may be several references to the same memory location within the program. Determining the set of all such references involves performing some form of *alias analysis*.

Both of these examples rely on some form of interpretation of the source language semantics – during folding the interpretation involved the semantics of the + operator and the semantics of integer representation, while during constant propagation the interpretation involved the semantics of the control flow constructs. Optimizers need to know about the underlying semantics of the language being transformed; it is

critical that the transformations performed by an optimizer are *semantics preserving*, i.e. that they don't change the meaning of the original program.

In some senses the techniques in this thesis are merely advanced versions of constant propagation and folding. We wish to use information which may be inferable from the source code for the purpose of answering various questions about the source. As one simple example, consider our `for` loop again. A naive interpretation would not be able to infer any knowledge about the state of the variable x following the loop. However, by inspection, it is clear that the value of x following the loop is going to be `15`. The techniques that we will be introducing are able to infer not only this information, but information that is much more general.

1.4 The Essentials of Abstract Interpretation

Fundamentally, interpretation should be understood as the implementation of semantics. In other words, an interpreter is a function whose domain (input) is a program in some language and whose range (output) represent the meaning of the program. In any programming language (or domain), semantic definitions are provided for expressions in the domain. These semantics may be given in varying degrees of formality – ML [63] being on the formal side and C++ [79] being on the informal side – but *all* languages give some sort of definition of the *meaning* of programs within the language. We will be using the term "standard semantics" to refer to the semantics defined for the original language.

Formally, we may express the meaning of a program as a function [] such that $[\![e]\!]$ is the "meaning" or interpretation of the expression e. The expression $[\![e]\!]_{\mathcal{L}}$ represents the meaning of the expression e when interpreted with the semantic definitions of language $\mathcal{L}$[1] – i.e. the behaviour of the expression e. Meaning functions may be specified in a variety of ways including denotational semantics, operational semantics, action semantics, or informal descriptions.

Using this notation we can more concisely describe a compiler. If c is a compiler written in language $\mathcal{L}$ which translates expressions from language $\mathcal{M}$ to some other

[1] When it adds to the clarity of the presentation, the subscript indicating the domain of the meaning function will be omitted.

language $\mathcal{M}'$ then the following equation should hold:

$$[e]_{\mathcal{M}} = [\,[c]_c(e)]_{\mathcal{M}'} \tag{1.1}$$

Intuitively, this says that the meaning of the expression in the source language should be the same as the meaning of the expression which results from compiling the expression. When such an equality holds we say the transformations applied by the compiler c are *semantics preserving*. Traditional compilers claim to be semantics preserving and are (more or less) accurate in their claims.

There are often circumstances in which an optimizer wishes to ask questions about a program in order to perform transformations. Such questions may include "is it possible for this segment of code to execute" or "can we determine the type of the object that is referenced by this pointer". The types of optimizations that rely on such questions include reachability analysis, live variable analysis, array partitioning, and interference computations for parallel applications. These types of analysis perform a crucial role in the optimization phases of compilation. It is useful to consider each analysis as an interpretation of the original program using a set of semantic definitions that is different than the semantic definitions of the original program. This permits a precise description of the method to be given and allows termination and performance characteristics to be established. *Abstract interpretation* is a general term which includes any such "non-standard" interpretation of expressions in a domain.

1.4.1 A Simple Example

One straightforward example of abstract interpretation is in determining whether the value of an arithmetic expression is negative, positive, or zero. Consider a language of mathematical expressions with addition, subtraction, and multiplication:

```
E :: E + F | E - F | F
F :: F * T | F / T | T
T :: (E) | constant
```

We can give a standard semantics for the language as the following:

$$\begin{aligned}
[\text{constant}] &= \text{constant} \\
[(e)] &= [e] \\
[e_1 + e_2] &= +([e_1], [e_2]) \\
[e_1 - e_2] &= -([e_1], [e_2]) \\
[e_1 * e_2] &= *([e_1], [e_2]) \\
[e_1 / e_2] &= /([e_1], [e_2])
\end{aligned}$$

This semantic definition gives the normal rules for evaluating expressions without dealing with the problem of division by zero. If we are only interested in whether the result is positive, negative, or zero we could define the following non-standard semantics:

$$\begin{aligned}
[\textit{constant}] &= \begin{cases} \text{neg if } \textit{constant} < 0 \\ \text{pos if } \textit{constant} > 0 \\ \text{zero if } \textit{constant} = 0 \end{cases} \\
[(e)] &= [e] \\
[e_1 + e_2] &= \oplus([e_1], [e_2]) \\
[e_1 - e_2] &= \ominus([e_1], [e_2]) \\
[e_1 * e_2] &= \otimes([e_1], [e_2]) \\
[e_1 / e_2] &= \oslash([e_1], [e_2])
\end{aligned}$$

Consider the following definitions:

$\oplus$	neg	pos	zero
neg	neg	???	neg
pos	???	pos	pos
zero	neg	pos	zero

$\ominus$	neg	pos	zero
neg	???	neg	neg
pos	pos	???	pos
zero	pos	neg	zero

In each of the abstract operations $\oplus$ and $\ominus$ there exist evaluations that do not have a "simple" answer composed of a single abstract value. For example, when a positive and negative number are summed, the result could be positive, negative, or zero. In general, many such situations can occur within abstract domains. For this particular case, we will allow subsets of the three basic abstract values to represent values. The abstract value *NPZ* will represent a set of abstract values composed of the *negative,*

positive and *zero* abstract values. The abstract operators must then be defined over all non-empty subsets of the abstract values. We will explicitly define the operators for single elements; the operators are defined to evaluate sets by taking the union of the results of applying the operation to all pairs in the cartesian product of the arguments.

The proper definitions for the abstract operators are then as follows:

$\oplus$	neg	pos	zero
neg	neg	NPZ	neg
pos	NPZ	pos	pos
zero	neg	pos	zero

$\ominus$	neg	pos	zero
neg	NPZ	neg	neg
pos	pos	NPZ	pos
zero	pos	neg	zero

$\otimes$	neg	pos	zero
neg	pos	neg	zero
pos	neg	pos	zero
zero	zero	zero	zero

$\oslash$	neg	pos	zero
neg	pos	neg	*error*
pos	neg	pos	*error*
zero	zero	zero	*error*

Using these non-standard semantics as the basis for an interpreter would result in an abstract interpreter for this language. Interpreting any expression in the language would result in a non-empty subset of the abstract terms *neg, pos* and *zero*. We would not know the actual result of the computation using the standard semantics, but we would have some abstract information about the expression.

Example 1:

```
(5 + (4 - 4)) ↦ (pos + (4 - 4))
(pos + (4 - 4)) ↦ (pos + (pos - 4))
(pos + (pos - 4)) ↦ (pos + (pos - pos))
(pos + (pos - pos)) ↦ (pos + NPZ) ↦ NPZ.
```

There is one important issue to note about this style of interpretation. By inspection, an accurate interpretation of the subexpression `4 - 4` should result in the abstract value `zero` rather than the abstract value `NPZ`. The process of abstraction has lost some of the information needed to reason accurately about the standard semantics. Any process of abstraction suffers from this problem to some degree; the key to a good system is to be flexible as to when information is lost. This theme will be re-addressed when the abstract domain requirements are introduced.

Example 2:

```
(5 - (-7 - 4)) ↦ (pos - (-7 - 4))
(pos - (-7 - 4)) ↦ (pos - (neg - 4))
(pos - (neg - 4)) ↦ (pos - (neg - pos))
(pos - (neg - pos)) ↦ (pos - neg) ↦ pos.
```

Although the previous abstract interpretation assumes that we have full knowledge about the values of constants, we can easily extend the model to admit "unknown" or "partially known" values. Admitting an *unknown* abstract value to the *neg*, *pos* and *zero* values changes only the abstract operators. Adding or subtracting values with an unknown value results in an unknown value. Multiplying or dividing with an unknown however, may result in a value other than unknown. Since we know that multiplying *any* number by zero generates zero, we can allow our $\otimes$ operator take advantage of operands which are *zero*. Division is similar, except that dividing anything by an unknown could result in an error since the unknown value might be zero. Incorporating *unk* as the abstract value for unknown within our model yields the following definitions for the abstract operators:

$\oplus$	neg	pos	zero	unk
neg	neg	NPZ	neg	unk
pos	NPZ	pos	pos	unk
zero	neg	pos	zero	unk
unk	unk	unk	unk	unk

$\ominus$	neg	pos	zero	unk
neg	NPZ	neg	neg	unk
pos	pos	NPZ	pos	unk
zero	pos	neg	zero	unk
unk	unk	unk	unk	unk

$\otimes$	neg	pos	zero	unk
neg	pos	neg	zero	unk
pos	neg	pos	zero	unk
zero	zero	zero	zero	zero
unk	unk	unk	zero	unk

$\oslash$	neg	pos	zero	unk
neg	pos	neg	*error*	unk, *error*
pos	neg	pos	*error*	unk, *error*
zero	zero	zero	*error*	unk, *error*
unk	unk, *error*	unk, *error*	unk, *error*	unk, *error*

Example 3:

```
(2 * (4 - ???))  ↦ (pos * (4 - ???))
(pos * (4 - ???))  ↦ (pos * (pos - ???))
(pos * (pos - ???))  ↦ (pos * (pos - unk))
(pos * (pos - unk)) ↦ (pos * unk) ↦ unk.
```

The `unk` value in the above is redundant. The behaviour of the `unk` value is exactly the same as the abstract value `NPZ`. This corresponds to intuition as well; an "unknown" value could be either `negative`, `positive`, or `zero`. Formalizing the properties of abstract domains makes this recognition more straightforward, even in complex domains.

These examples illustrate the basic method for defining any abstract interpretation: define an abstract domain (set of abstract values), define the operators on those values, and define the method for applying those operators to expressions in the language. Most of this work will focus on the first two of these requirements; the third will follow in a fairly natural way from the domains and operators we define.

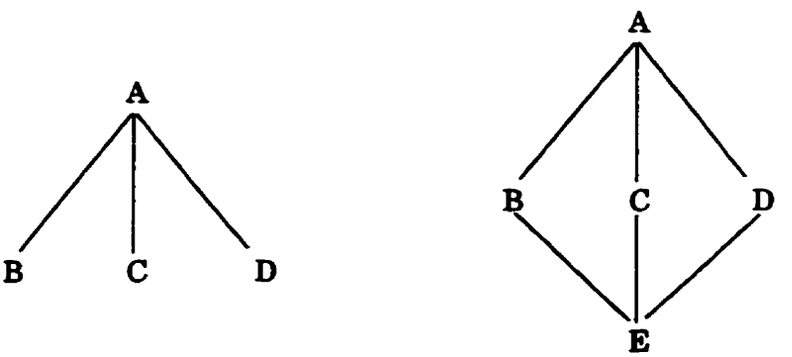
1.5 Formalizing Abstract Relationships

The standard models used for formalizing abstract domains are developed from lattice theory. In this section, we briefly review some notation and the basics of lattice theory; more detail is contained in Appendix A, but for a complete development, we would recommend the introductory book by Davey and Priestley [30].

A *lattice* is a formal model for describing the relationships between elements in a set. A lattice is a special case of a *partial order*.

Defn 1.1 (Partial Order) *A* partial order $< S, \preccurlyeq >$ *is a set* S *and a relation,* $\preccurlyeq$*, on* S *such that for* $x, y, z \in S$*, the* $\preccurlyeq$ *relation is:*

- **transitive:** $x \preccurlyeq y$ *and* $y \preccurlyeq z \Rightarrow x \preccurlyeq z$.
- **antisymmetric:** $x \preccurlyeq y$ *and* $y \preccurlyeq x \Rightarrow x = y$.
- **reflexive:** $x \preccurlyeq x$.



(i) Lower bounds {B C D}
No Greatest lower bound

(ii) Greatest lower bound {E}

Figure 1.5.1: Lower bounds

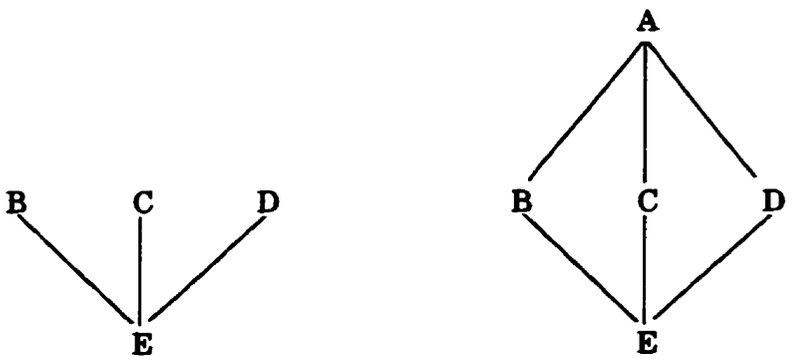
If $x \preccurlyeq y$ we may say that x is *below* y. Note that it may be the case that $\preccurlyeq$ does not hold at all between two arbitrary elements of S. In other words, it may be the case that for some $x, y \in S$, $x \not\preccurlyeq y$ and $y \not\preccurlyeq x$. In such a case we say that x and y are *incomparable*, denoted as $x||y$.

It is useful to be able to talk about various *bounds* or limiting values of a subset of some partial order $<S, \preccurlyeq >$. Assume that S' is a subset of S for some partial order $<S, \preccurlyeq >$.

Defn 1.2 (Lower Bound) *A* lower bound *for S' is an element $y \in S$ such that $\forall x \in S', y \preccurlyeq x$.*

Note that the lower bound of a subset of S does not have to be a member of the subset, it is only required to be a member of S.

Defn 1.3 (Greatest Lower Bound (GLB)) $\sqcap S'$, *the* greatest lower bound *for S' is a lower bound, y, of S' such that $\forall x \in$ { lower bounds of S'}, $x \preccurlyeq y$. We will also refer to the greatest lower bound of a set containing elements x and y as the* meet *of x and y, denoted as $x \wedge y$.*



(i) Upper bounds {B C D}
No least upper bound

(ii) Least upper bound {A}

Figure 1.5.2: Upper Bounds

It may be the case that a set does not have a lower bound; if there are two incomparable values which constitute the set then there would be no value comparable to (and below) every element of the set.

Defn 1.4 (Upper Bound) *An* upper bound *for S' is an element $y \in S$ such that $\forall x \in S, x \preccurlyeq y$.*

Defn 1.5 (Least Upper Bound (LUB)) *$\sqcup S'$, the* least upper bound *for S' is an upper bound, y, of S' such that $\forall x \in \{$ upper bounds of $S\}$, $y \preccurlyeq x$. We will also refer to the least upper bound of a set containing elements x and y as the* join *of x and y, denoted as $x \vee y$.*

As with lower bounds, an upper bound may not exist. Obviously it is the case that if no upper bound exists, no least upper bound exists.

Defn 1.6 (Lattice) *A* lattice $< S, \preccurlyeq >$ *is a partial order such that $\forall \{x, y\} \in S : x \wedge y$ and $x \vee y$ exist.*

A lattice requires that a least upper bound and greatest lower bound exist for any pair of elements. The nature of these bounds has some relationship to minimality; the least upper bound for two elements is the "smallest" value that represents both of the values.

A lattice may have distinguished elements, labeled $\top$ (top) and $\bot$ (bottom), that represent the greatest and least elements in the lattice. Any finite lattice will have such elements; infinite lattices may not.

Chapter 2

Partial Evaluation and Symbolic Execution

Partial evaluation and symbolic execution are general terms which encapsulate methods for more complete forms of abstract interpretation. The goal of partial evaluation is to interpret programs in which only part of the input data is known at interpretation time. Given an interpreter *int* in language $\mathcal{L}$ and data d, a traditional interpretation for a program e may be expressed as:

$$[int]_{\mathcal{L}}(e, d) \tag{2.1}$$

Partial evaluation considers the data as being composed of two distinct parts – a *static* part and a *dynamic* part. The static part of the data contains information which does not change between interpretations of the program. The dynamic data contains the information which is not available until the program actually runs. This view of the data is reflected in the following:

$$[int]_{\mathcal{L}}(e, s, d) \tag{2.2}$$

where s is the static portion of the data and d is the dynamic portion of the data. Note that this equation factors the static data out of the entire set of program data.

2.1 The *Mix* Equation

Partial evaluation arises from the recognition that equation 2.2 can be rearranged to incorporate the static data, s, into a new program that, when applied to the dynamic data d, provides the same results as the original program. A partial evaluator, *mix*, from a language $\mathcal{L}$ to a language $\mathcal{M}$ takes a program e in $\mathcal{L}$ and data s and produces a new program e' in $\mathcal{M}$ such that the following holds:

$$[\![e]\!]_{\mathcal{L}}(s,d) = [\![e']\!]_{\mathcal{M}}(d) \tag{2.3}$$

Expanding e' into its symbolic form we get the following:

$$[\![e]\!]_{\mathcal{L}}(s,d) = [\![[\![mix]\!](e,s)]\!]_{\mathcal{M}}(d) \tag{2.4}$$

Equation 2.4 is called the ***Mix Equation***. The name *mix* stems from work by Ershov [32] on *mixed computation* which was pioneering work in computation with mixed dynamic and static data. An early partial evaluation system [48] was called *mix* in recognition of this contribution. Although the term *mixed computation* has been superseded by the term *partial evaluation*, the name *mix* has been retained as the common name for symbolic interpreters.

The Mix Equation is interesting in that it reflects the same basic process as currying in functional programming [54]. Currying occurs in functional languages when functions with multiple arguments may be viewed as functions that take a single argument and return a function over the remaining arguments. Currying can be expressed formally through the lambda calculus as nested function definitions (lambda expressions). Similarly, a partial evaluator views a program as taking static data and returning a program over the dynamic data.

2.2 The Futamura Projections

Let us briefly restrict the general form of partial evaluators to consider only partial evaluators whose target language is the same as the source language. That is, let *mix* be a partial evaluator from $\mathcal{M}$ to $\mathcal{M}$. In addition, let *int* be an interpreter written in language $\mathcal{M}$ which interprets programs in $\mathcal{L}$.

$$r = [\![mix]\!](int, e) \tag{2.5}$$

Consider the result, r, of this application of *mix*. Applying r to static and dynamic data will give the same result as interpreting e with the same data. That is,

$$[e]_{\mathcal{L}}(s,d) = [int]_{\mathcal{M}}(e,s,d) = [[mix](int,e)]_{\mathcal{M}}(s,d) = [r]_{\mathcal{M}}(s,d) \quad . \tag{2.6}$$

The critical observation is that r has the same behaviour as a *compiled* program. This identity, called *The First Futamura Projection* [34], shows that *mix* may be used to generate a compiled program from an interpreter and a source program.

In equation 2.4 and equation 2.5 we did not define the source language of the partial evaluator, *mix*. Let us now assume that *mix* is itself written in language $\mathcal{M}$. We can push the level of interpretation out an additional level.

$$comp = [mix](mix, int) \tag{2.7}$$

This makes more sense when you consider applying a program, e, and data (s,d).

$$[\,[comp]\;e]\;(s,d) = [\,[[mix](mix,int)]\;e]\;(s,d) \tag{2.8}$$

comp is a compiler – it takes a program, e, and produces a program which, when applied to the data, produces the result of the original program. Using this approach, *mix* can automatically produce compilers from interpreters. This level of application is called *The Second Futamura Projection*.

The Third Futamura Projection pushes the application of *mix* out one more level. Consider the following:

$$cogen = [mix](mix, mix) \tag{2.9}$$

Again, consider applying the rest of the arguments to cogen:

$$[\,[\,[[mix](mix,mix)]\;int]\;e]\;(s,d) \tag{2.10}$$

cogen acts as a compiler generator. Given an interpreter, *cogen* produces a compiler which may be used as described in the Second Futamura Projection.

The Futamura projections rely on having partial evaluators which are written in the same language as the language of interpretation. Such partial evaluators are called *self-applicable*. There is continuing work in self-applicable partial evaluators with the view towards automating compiler production for realistic environments. There are a number of difficulties with this approach; it is difficult to see how to

automatically map data layout in an interpreter to data layout in compiled code and it is unclear whether a partial evaluator could "discover" data relationships which could be transformed into data structures which don't exist in either the partial evaluator or the interpreter. [16] [43] [45] [55] [66] [84]

There are other issues for automatic compiler generation with respect to efficiency, code generation, and other low-level machine specific requirements. Although the Futamura Projections are interesting and continue to spur research, the remainder of this document will not deal with self-application issues. The focus will be not on automatic compiler generation, but rather making use of the underlying techniques to discover information that could be used in a somewhat more traditional compilation system.

2.3 General Concepts of Partial Evaluation

2.3.1 Specialization

At their core, the Futamura Projections express the idea of *specialization* – the incorporation of specific data into a general program for the purpose of generating a specialized version of the program. If we consider the Third Projection as a basis for expressing computation then we can express any program behaviour as a specialization of some instance of *cogen*.

General forms of partial evaluation incorporate specialization as a fundamental aspect of their behaviour. A *specialization* occurs when a partial evaluator integrates some piece of static data into a code fragment and produces a new code fragment. A *specializer* performs specializations based on whether a particular value is static or dynamic. In most systems, *annotations* are introduced into a program which mark a value as static or dynamic. The process by which such annotations are introduced is called *binding time analysis* and will be discussed at length beginning in Section 2.3.2.

Consider the following simple function, f, and a call to the function:

```
(define f
    (lambda (x y)
        (+ x y)
))

(f 3 z)
```

Assuming that we do not know the value for z, the program is annotated as:

```
(define f
    (lambda (x y)
        (+ x_s y_D)
))

(f_D 3_s z_D)
```

where x_D means that x is dynamic and x_s means that x is static. Both function calls and variables may be annotated. The annotation on a function call reflects whether the function will be entirely evaluated (is static) or have a function call left in the specialized program (is dynamic). For a variable, the static annotation means that the specializer may use the value during the specialization while the dynamic annotation means that the specializer may not use the value.

Specialization may be an identity operation – the specializer may not have enough static information to perform a specialization, or the specializer may not be allowed to perform a specialization even though some static data is present. The latter case occurs in some special situations which will be discussed in later sections. In all cases, the result of a specialization is called a *residual*.

Given the above annotations, a residual for our program might be as follows:

```
(define resid-f-1
    (lambda (y)
        (+ 3 y)
))

(resid-f-1 z)
```

The fundamental algorithm for a partial evaluator consists of selecting a function for specialization, producing a residual through some specialization and repeating until all useful static information has been used. Figure 2.3.1 gives one basic algorithm for specialization.

```
fun spec (code, actual)
   for each sequential line of code
     for each operator or function do (in evaluation order)
       if a function call
         replace by the residual from spec(function,arguments)
       else if an operator and arguments have static values
         replace by the result from evaluating operation
   return remaining code as residual
```

Figure 2.3.1: Basic Specialization

2.3.2 Binding-Time Analysis

As noted earlier, partial evaluation techniques consider program data as being in one of two classes – static or dynamic. Although we expressed both sets of data as parameters to the program, in reality a great deal of static information may be present in the text of the source program alone. This part of the program data must also be considered as static and can conceptually be considered as part of the parameters to the program (a simple rewriting could be performed in order to have such data presented as parameters, but it is not necessary to do so). When we consider performing partial evaluation on a program, one of our first concerns will be to decide which of the program variables we will want to treat as containing static data and which we will have to treat as containing dynamic data. The process by which data is divided between the two classes is called *binding-time analysis* [54] [46]. Normally a binding-time analysis will introduce annotations into a program to represent the status of each variable. These annotations are then used by the specializer to determine what information may be incorporated into the residual program.

There are several issues involved in binding-time analysis (BTA): *termination*, *accuracy*, and *lifetime*. Binding-time analysis is in general not decidable, so all techniques must approximate the actual set of static and dynamic data within the program. The calculation of a reasonable estimate involves iteratively making an estimate and then checking whether some type of fixed-point has been reached within a solution set. The BTA process must terminate while not making an overly conservative approximation in order for the information to be useful within the specialization phase. The termination problem is also referred to as the problem of *divergent com-*

putation or simply *divergence*.

Accuracy relates to the "resolution" of the analysis. The simplest approach is to have a single annotation for each variable. This approach is generally not very accurate since a single variable may name a compound data structure, some of which may be dynamic and some of which may be static. A more accurate analysis involves treating each member of a compound data structure as a distinct binding by associating a binding-time annotation with each of the elements. Issues which affect the accuracy of the analysis include the memory model, the presence of higher-order structures, and aliasing and side-effect mechanisms. Increased accuracy provides more information to the specializer at the cost of increased computation time and more sensitive termination criteria. There are two restrictions on any BTA: the BTA must be *safe* in that no dynamic expression may be annotated as static, and the BTA must be *useful* in that all static expressions (or at least as many as possible) are denoted as static [73]. Both safety and usefulness effect termination and accuracy.

Finally, the lifetime aspect of a binding-time analysis relates to whether there is a single annotation for a variable or if there may be several annotations which apply at different points within the program. Most current techniques have only a single annotation for each variable, although there is continued research into techniques which allow for multiple annotations. Lifetime decisions relate to the interaction of the binding-time analysis and the specialization phase; this interaction is the topic of Sections 2.3.5 and 2.3.5.

2.3.3 Types of Partial Evaluators

There are four fundamental approaches to partial evaluation. These approaches combine one of two methods for lifetime analysis with one of two methods for the relationship between the BTA and the specializer. The two lifetime methods are *monovariant* and *polyvariant*; the two relationships are *on-line* and *off-line*. The following sections will discuss each of these methods. In addition to the above classifications, there are a number of orthogonal issues. Memoization (Section 2.5) and accuracy are two of these issues that we will discuss.

2.3.4 Polyvariant and Monovariant BTA

Monovariant BTA

The main issue concerning lifetime considerations is whether the binding time analysis (and the resulting specializations) will be monovariant or polyvariant. A *monovariant* analysis generates a single set of annotations for a particular segment of code (usually a function). These annotations are then used for the entire specialization phase [11].

The fundamental problem with monovariant BTA lies in the fact that there is only a single annotation for each variable in a function. The annotation for a particular variable must then be the *most general* (or *widest* [73]) annotation for any possible run-time binding of values. For a given formal parameter, if there exists a call site in which the actual parameter is static and another call site at which the actual parameter is dynamic, then the annotation for the formal parameter will be dynamic and the specialization phase will not be able to make use of the information available in the static parameter.

We will again use our simple function, `f`, as an example.

```
(define f
    (lambda (x y)
        (+ x y)
))
```

Assume that we have the following calls of `f`:

```
(f 3 z)
(f z 3)
```

with `z` being *dynamic*. The annotated version of `f` and the uses of `f` would be as follows:

```
(define f
    (lambda (x y)
        (+ x_D y_D)
))

(f_D 3_S z_D)
(f_D z_D 3_S)
```

Figure 2.3.2: Lattice of Simple Annotations

For each parameter there exists a call with a *static* actual argument and a call with a *dynamic* actual parameter. The most general annotation for each parameter is *dynamic*, so the annotations for both variables in the function body become *dynamic*. As a result of these annotations, no specialization will be performed during the specialization phase and the original function will remain as the residual.

The two annotations, *static* and *dynamic*, form a very simple lattice as shown in Figure 2.3.2. Note that we do not show a $\top$ or $\bot$ element in the lattice. The two elements in the lattice actually have the correct properties for $\top$ and $\bot$ so we do not need the additional elements. Alternatively, the lattice could be seen as containing *only* $\top$ and $\bot$ with the renaming of $\top$ to D and $\bot$ to S.

The monovariant approach can be clarified using this simple lattice. Monovariance uses the least upper bound of the annotations at all call sites as the annotation for a function. [46] expresses this by using an analysis function $\mathcal{B}_v$ which takes a binding time environment (set of annotations), a function g, and an expression, e. The result is the least upper bound of the annotations for g within e. You can then express g's monovariant annotation as:

$$\bigsqcup_{i=1}^{n} \mathcal{B}_v\,[e_i]\,\tau\,g \tag{2.11}$$

where τ is the least upper bound of annotations for all other functions and e_i is the i^{th} expression in the program. Given this definition, if *any* $\mathcal{B}_v$ annotation results in D (dynamic) as the annotation for a parameter to a function, the least upper bound will necessarily be D.

The monovariant approach does not provide the generality needed for most realistic applications of partial evaluation. In real programs it is unlikely that *all* call sites for a particular function will have a static value for any given parameter. A great deal of static information is ignored in a monovariant approach, decreasing the effectiveness of the entire partial evaluation process.

Polyvariant BTA

Polyvariant techniques differ from monovariant techniques in that different annotations can be made at every function call site. If there is a call site in which an actual parameter is static and another call site in which the actual parameter is dynamic, then two sets of annotations would be made. In order to make two sets of annotations for a single function, the source function is simply duplicated[1]. [73]

```
(define f
    (lambda (x y)
        (+ x y)
))
```

Again assume that we have the following calls of f:

```
(f 3 z)
(f z 3)
```

with z being *dynamic*. For the first call, we shall produce an annotation with the first parameter *static* and the second parameter *dynamic*. For the second call we will produce a complementary annotation. Conceptually, we have the following functions and annotations after the BTA phase:

```
(define f-1
    (lambda (x y)
        (+ x_S y_D)
))

(f-1_D 3_S z_D)
(f-2_D z_D 3_S)
```

```
(define f-2
    (lambda (x y)
        (+ x_D y_S)
))
```

During specialization, the following functions and calls will be produced:

```
(define resid-f-1
    (lambda (y)
        (+ 3 y)
))

(resid-f-1 z)
(resid-f-2 z)
```

```
(define resid-f-2
    (lambda (x)
        (+ x 3)
))
```

[1] Rytz et al do not in fact duplicate the actual source code, but rather keep multiple annotations for each function.

The formal lattice model does not change in the polyvariant approach. The difference in approaches is due to the application of the model; we no longer use the least upper bound of a set of annotations, but rather introduce sets of annotations for each function. Since each annotation has only two possibilities (static or dynamic) and each parameter list is finite, we have a finite number of possible annotations.

Clearly this approach is superior to the monovariant approach – if *any* call site has static information which can be used, an occurrence of the function with a useful annotation will exist. The obvious problem is that there may be many annotations for a given function. The number of potential annotations is bounded by the number of call sites, but could be exponential in the number of parameters if many call sites exist. Exponential growth of the residual code during specialization is a problem for any polyvariant approach and will be discussed in later sections.

There is however a subtle problem in the way in which polyvariant BTA is normally used. Conceptually, polyvariance creates instances of the functions which are being annotated, but this is exactly what the specializer is supposed to be doing. Polyvariant analyzers duplicate some of the work which is to be done during specialization if the BTA is performed strictly before the specialization phase. With polyvariant analysis, it seems to make more sense to combine the BTA and specialization phases into a coherent approach. The comparison between separate BTA and combined BTA/specialization is the topic of the next section.

2.3.5 Off-line and On-line Approaches

Off-line Techniques

Off-line BTA techniques analyze the source program before the specialization phase and determine the status of each variable. Each variable and function call is annotated as being either static or dynamic. During the specialization phase, dynamic values are never specialized, while static values are always specialized.

There are a number of techniques for off-line BTA[2]. The most common techniques are based on type inference or constraint analysis algorithms. Newer techniques [54] employ projections to create annotations. The type inference approach incrementally adds dynamic notations until the inference algorithm succeeds. Constraint based

[2]For a more detailed introduction to these techniques, see [46].

systems generate constraints, convert them into a normal form and then solve the constraints to generate mappings from variables to annotations. Brief discussions of some of these approaches are given in Section 2.4.3 when we discuss termination issues.

On-line Evaluation

In on-line evaluators, the decisions regarding effective annotations are interleaved with specialization decisions. At each step in the evaluation an on-line evaluator must decide what to treat as dynamic and what to treat as static. For a given function or variable this decision is independently made every time that the function or variable is encountered. Once the decision is made, the specialization takes place immediately and the residual becomes part of the next set of evaluations. In a sense, on-line partial evaluation is naturally polyvariant since the algorithm itself "reconsiders" decisions on a frequent basis.

There has been relatively little work in the area of on-line partial evaluation; the most significant implementation work has been done by Katz, Weise, and Ruf in the FUSE evaluator [83] [71][70]. Although they had an interesting approach for dealing with redundancy in specializations, their model for values was not very expressive. Termination in FUSE relies on having a finite height lattice modeling values in the system. In addition, there are circumstances in which FUSE requires user provided "finiteness annotations" that guarantee that specialization will terminate. Although such annotations allow FUSE to incorporate more selective residual production algorithms, such annotations require user intervention. The automatic on-line approach that we will be presenting will incorporate features similar to their approach but allows infinite height lattices to model values in that system and will feature a clear separation between models for abstract values and the algorithm itself.

The most significant approach that formally proves more of the properties of on-line partial evaluation is the work by Consel and Khoo [25][24]. This work will be discussed in Section 4.5 after we have developed the basis of our system.

Combinations of Methods

Table 2.3.1 shows which of the four possible approaches have been investigated. Monovariance will be discussed in the next section, and the two polyvariant meth-

	Monovariant	Polyvariant
Off-line	Similix	Similix-2
On-line	unknown	Our approach FUSE

Table 2.3.1: Types of Partial Evaluators

ods will be discussed in some detail in later sections. As noted in the table, early approaches, such as Similix, used an off-line monovariant approach, while the more sophisticated Similix-2 uses an off-line polyvariant approach. As noted above, there has been relatively little work in on-line approaches. The most substantial work is a polyvariant on-line interpreter by Weise et al [83]. At this time, we do not know of any on-line monovariant approaches.

2.4 Other Issues

2.4.1 Higher-Order Languages

The fundamental problem when dealing with higher-order languages lies in finding all potential call sites for a higher-order function. When we do not know whether parameters will be dynamic or static, we must assume that they are completely dynamic (due to the safety constraint). This is the approach taken by most systems, such as Similix-2 [11], Schism [21] and FUSE [83]. A more accurate knowledge of higher-order function analysis relies on some form of control-flow analysis, such as in [72] (see [75] for control flow analysis techniques). The control flow analysis is used to create a conservative estimate of the call sites and then uses the least upper bound of the argument annotations as the annotation for the function parameters.

A control flow analysis certainly improves the accuracy of the annotations, but requires significantly more work. In addition, the calculation of the estimation involves much of the same type of analysis as used in a specializer. The control flow analysis in [72] does not create polyvariant residuals for higher order functions, but acts as a monovariant specializer for the function. In any finite program, however, there will be a finite number of higher order functions which could be used to create more accurate, polyvariant residuals if the specializer tracked the set of possible higher-order

functions that could be used at any call site. Such an analysis reverses the control flow analysis approach; rather than estimating the call sites for each higher-order function, such an approach would estimate the domain of each call site. One could then specialize each higher-order function in the domain with the static information available at the call site. In some ways such an approach would entail a much more complete control flow analysis, but if integrated with the specializer, would be no more costly.

2.4.2 Languages with Imperative Features

The major problem with off-line binding-time analysis is that the techniques assume that annotations do not change during the specialization phase. Although this is normally the case in (pure) functional languages, this assumption breaks down in the face of imperative features. Consider the following imperative code:

```
read(x);
y := x + 5;
x := 7;
```

In this example, x is dynamic after the read, but becomes static after the assignment in the third line. With imperative languages the status of a variable can change at any time due to either a direct assignment or an assignment to an alias for the variable. This cannot be reflected in off-line techniques [62] which do not incorporate any idea of a change of use into the binding-time analysis.

Some experiments have been conducted [5] with C that attempt to use an off-line BTA. One of the major problems is in dealing with dynamically allocated arrays which are then assigned static values. Due to the dynamic nature of the array allocation, normal off-line BTA would treat the entire array as dynamic and miss many opportunities for specialization. The approach taken in this work was to convert the dynamic array into a static array which can then be analyzed more accurately by traditional off-line techniques.

In many cases the dynamic allocation to static allocation transformation would not pose any problems in the residual program. Unfortunately, this transformation is not strictly semantics preserving due to the underlying memory model of the C language. This could introduce problems in situations where stack or static data space is severely limited; one such example is in threads-based programming support packages such as the μSystem [15] developed at the University of Waterloo.

Relationship to SSA

Static Single Assignment (SSA) [82] [28] is an abstract interpretation approach to high-performance Fortran optimization problems. SSA converts a source program into one in which each variable may be expanded into several instances of the original variable. Each assignment statement to the variable creates a new instance of the variable, and any use of a variable is converted to a use of the appropriate instance of the variable. This approach guarantees that there is a unique assignment instance for each variable at any point within the program.

For example, the following Pascal-like code:

```
read(a);
b := a + 5;
a := 7;
c := a * 5;
```

would be converted using the SSA approach into the following:

```
read(a_1);
b := a_1 + 5;
a_2 := 7;
c := a_2 * 5;
```

The SSA community uses the SSA transformations to do fairly straightforward types of abstract interpretation – code is specialized based on the values of the appropriate instances of variables. SSA does not attempt to create new specialized instances of any code, and as such resembles the monovariant approach discussed in Section 2.3.4. The interesting aspect of this work is in its correlation to off-line BTA, and in its approach to imperative features. Combining the SSA transformation with the polyvariant features of off-line partial evaluation would generate an approach which is more powerful than either in isolation; such a system would be able to deal with changing annotations after assignments since each assignment would have a different instance of the variable associated with it.

Doing a "normal" annotation for the first code fragment would result in the variable `a` being annotated as *dynamic* since the result of the `read` is not known until run-time. Since there is only one annotation for a particular variable within a section of code, we would have to use *dynamic* and would lose the static information which occurs within the same section of code. Using the SSA converted code however, we

would have two distinct variables from the original a. Each of these would receive the appropriate annotation, allowing us to make use of the static information from the second assignment to a.

There are a few potential difficulties with combining the SSA and off-line approach however. First of all, the SSA literature has not addressed imperative languages such as C in which there are arbitrary aliasing relationships. Such aliasing would complicate the SSA conversion to such an extent that it probably would not be viable for realistic systems. Second, the difficulties in generalizing off-line methods to higher-order constructs is not alleviated by the introduction of the SSA conversion. Call locations which are truly dynamic would not be resolved by SSA; a more general domain model is required. Finally, this approach would retain the duplication of work mentioned in Section 2.3.4. The effect of SSA conversions during a binding time analysis can be achieved by adopting more general on-line approaches such as our approach.

2.4.3 Termination

Off-line Evaluation

Termination for off-line partial evaluation depends entirely on the termination and safety of the binding time analysis. When off-line methods are used, specialization blindly follows the annotations on the variables and does not check for any special termination conditions. The BTA has the responsibility for ensuring that the specializer will not attempt a recursive specialization which will not terminate. For example, given the following code:

```
(define sum
    (lambda (start, stop)
        (if (= start stop)
            start
            (+ start (sum (+ 1 start) stop)))
))
(sum 5 z)
```

assume that z is *dynamic*. Even though the BTA knows that start is *static*, it should not annotate it as such. If start were annotated as *static*, the specializer would create a specialized version of sum as follows:

```
(define resid-sum-1
    (lambda (stop)
        (if (= 5 stop)
            5
            (+ start (sum 6 stop)))
))
```

Using the same argument, the recursive call to sum would cause another specialization of sum resulting in:

```
(define resid-sum-2
    (lambda (stop)
        (if (= 6 stop)
            6
            (+ start (sum 7 stop)))
))
```

Continuing with this, it is clear that the specializer would create an infinite number of residuals for sum. To avoid this problem, the BTA is required to make *safe* annotations — annotations which guarantee that infinite specialization does not occur. There are a number of approaches that have been investigated for performing safe binding time analysis; examples include constraint satisfaction [41] [42], type inference [35], program factorizations [54], and simple abstract interpretations [46].

The type inference approach is interesting in that BTA questions can be answered by giving static values their known types and then using type inference to determine resulting types. Annotations are progressively relaxed until the program is well-typed. Any well-typed expressions can safely be treated as static since they would depend only on other static values. For example, consider our prior sum example. Initially setting `start` to a known type (integer) and then performing type inference will result in a type conflict between `start` and the inferred type of "=" since the type of `stop` is unknown. Thus the type of `start` would have to be modified to unknown in order to resolve the conflict.

Another approach, based on projections, is suggested by Launchbury [54]. The foundation of projection based approaches is in normal set projections (a form of retractions, or subset selection). When we consider the set of parameters to a function, we need to find a projection of the parameters which represents the static parameters. The complementary projection would then provide the dynamic parameters

to the function. Launchbury's Ph.D. thesis [54] summarizes previous work in this approach and develops a formalized, extended model for BTA projections.

On-line Evaluation

The problem of divergence becomes somewhat more difficult using on-line approaches. Again, consider the `sum` example. When an on-line algorithm initially evaluates the function it will not be able to fully determine the value of the conditional since `stop` is dynamic. The algorithm then has a few choices about how to proceed. One choice would be to "give up" on the evaluation and simply to produce the original function as the residual. It should be clear that this is not a reasonable option since following this choice would mean that no function with *any* dynamic characteristics would be specialized. The interpreter cannot arbitrarily choose to use only one of the branches since such a choice would change the possible run-time behaviour of the resulting program. The final option is to investigate both branches of the expression and to build a residual based on both investigations. Unfortunately, blindly applying the third option introduces divergence if there exist any possible circular recursions within the original program, as occurs in the code for `sum`.

The general approach taken by on-line partial evaluators is to follow a combination of the first and third options. There are several issues involved in deciding on the exact method — the accuracy of the model for static and dynamic values, the allowable size expansion in the final residual, and the time spent performing the evaluation. Katz and Weise [83] have an interesting approach to accuracy concerns. In their system they incorporate an explicit "use-analysis" to determine which values are used to control recursive evaluations. The variables that are used to control recursive evaluations are subject to fixed point calculations while independent variables are allowed to "grow" in an arbitrary way. Ruf [70] presents informal arguments for the termination of the FUSE on-line system, but does not formally prove termination. As we progress through the development of our approach in Chapter 3, we will discuss these issues in more detail.

2.5 Residual Code and Specialization

Within a particular code fragment there may be many calls to a particular function. Each of these calls may have a different partitioning of static and dynamic data and the static data which is present may differ from call to call. Partial evaluation as described so far specializes a function each time an instance of the function is encountered. Following this approach blindly, however, is not very efficient as there may be many identical residuals produced.

Recall the simple function, `f`, used in earlier sections:

```
(define f
    (lambda (x y) (+ x y))
)

(f 3 x)
(f 3 y)
```

In this case we have two calls to `f` in which the first argument has the static value 3 and the second argument is dynamic. If we perform a simple specialization for this calls, we generate two identical residuals.

```
(define resid-f-1
    (lambda (y)
        (+ 3 y)
))
```

```
(define resid-f-2
    (lambda (y)
        (+ 3 y)
))
```

Clearly it is advantageous to have only a single instance of any particular residual. Specializers typically do this by *memoizing* residuals as they are produced. Memoization is simply a form of caching for residual code; memoization associates all static information used in the specialization with the corresponding residual. The issue of memoization is orthogonal to the issue of choosing on-line or off-line evaluation; in either case we wish to eliminate the production of duplicate residual functions.

Memoization can be seen as a mapping from a source code fragment and environment to a residual code fragment. If ρ is an environment mapping identifiers to values, then memoization is a mapping, $\mathcal{M}$, such that $\mathcal{M}(f, \rho) \rightarrow r$ where r is a version of f specialized with values from the environment ρ. Figure 2.5.1 shows how function `f` and an environment with `x` having the known value 3 maps to the residual `resid-f-1`. Before a specializer produces a new residual r' for a function f in

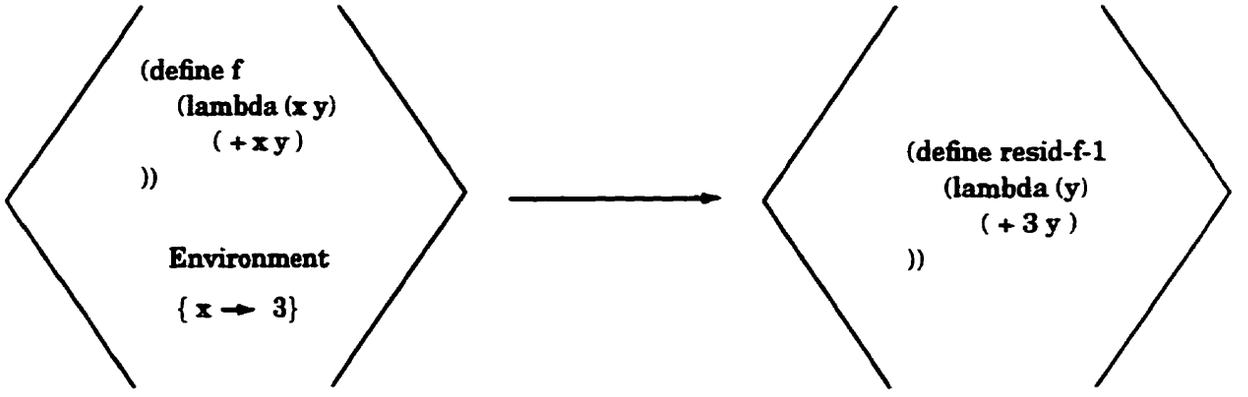


Figure 2.5.1: Memoization Map

an environment ρ', it checks the memoization map for a mapping which matches the current function and environment. More formally,

$$r' = \begin{cases} \mathcal{M}(f, \rho') & \text{if } \mathcal{M}(f, \rho') \text{ is defined} \\ \text{specialize}(f, \rho') & \text{otherwise} \end{cases}$$

If a residual exists for a given function and environment that residual is re-used, otherwise a new residual for the function is produced.

Memoization creates equivalence classes between residual functions and represents each class as a single residual. Membership in a particular equivalence class is determined by the function and the environment in which specialization occurs. Viewing residuals as existing within equivalence classes encourages a clean model for overall code expansion; if the total number of equivalence classes grows beyond the expansion limit, equivalence classes must be combined into larger classes.

Combining two equivalence classes involves creating a new class whose arguments are annotated with an annotation that covers the annotations in the classes being combined. Memoization itself, however, is orthogonal to the different types of partial evaluation, and combining equivalence class annotations cannot be expressed as movement in the simple lattice given in Figure 2.3.2. For example, consider the residuals given in Figure 2.5.2. Each residual was created from the original function with x marked as static. Strictly from the lattice, the least upper bound of the two is still static but this does not reflect the fact that two different values for x were used in the specializations.

```
(define resid-f-1
    (lambda (y)
        (+ 3 y)
))
```

```
(define resid-f-2
    (lambda (y)
        (+ 4 y)
))
```

Figure 2.5.2: Two different residuals

We could formally express memoization by using a disjoint union of the annotation lattice and a lattice of constant values (such as the integer lattice in Figure A.3), but as memoization is a small issue with respect to the techniques presented here, we will ignore the formal mechanism. Intuitively however, coalescing two equivalence classes means that two functions with the same static property will retain the static property in the resulting class, whereas if the annotations or static values are different, the result will become dynamic, or will at least lose some of the precision of the derived knowledge about the value.

2.6 Applications of Partial Evaluation and Specialization

2.6.1 Reducing Costs of Polymorphism

Object Oriented Languages

An important aspect of any object oriented language is the separation between a *message* to an object and the *method* that is used to respond to the message. The particular method that is used to respond to a given message is dependent on the object to which the message was sent. The term *dynamic binding* is used to refer to this run-time binding of messages to methods. In general, the encoding of each object will need to incorporate references to the methods that are used for its messages. Often languages will only allow the message to method association to be changed on a class-wide basis, but in either case, every invocation of a method will require at least one additional level of indirection.

The choice of a method cannot generally be made at compile-time due to inheritance polymorphism. Inheritance polymorphism requires that any subtype object can be used in the place of a base type object. This means that in general it is undecidable whether a given method is used for a given message invocation. In [31], Dean

et al investigate applying specialization techniques to reduce the cost of method call. The basic idea is to introduce specialized instances of methods, where each version is specialized with respect to particular subtypes of the allowed parameter type. For example, if a given method allows a single parameter of a type A and there exists subtypes B and C of A, then instances of the method could be created for each of A, B and C. These methods could then take advantage of the fact that the actual type of the object is known and could potentially create static bindings for the following messages. Generally however, the number of potential methods that *could* be generated is much larger than the number that *should* be generated.

Dean et al use a call-graph based estimation algorithm to estimate which object specializations are likely to be profitable. In order to determine the effect of selective specialization, they incorporated their technique into the Cecil [18] compiler and self-compiled the compiler. The overall result was quite impressive; at the cost of an increased program size of only 5%, run-time improvements of approximately 33% were achieved.

This result is very encouraging for the future application of partial evaluation techniques. The work by Dean et al was based on a fairly simple call analysis and did not attempt to perform any analysis of the form that traditional off-line BTA performs, nor apply any of the on-line evaluation techniques. They did, however, clearly demonstrate that moderate code size increase in a polymorphic environment can lead to substantial performance improvements.

An approach more closely tied to partial evaluation techniques is the work by Khoo and Sundaresh [50]. Their approach was based on using continuations[3] which allow for similar types of re-associations of methods and messages. Harnett and Montenyohl [38] have also investigated continuations and caching based approaches in an object oriented language. Finally, Marquard and Steensgaard [59] have developed an automatic on-line approach that uses similar techniques as applied in FUSE.

Run-time Overload Resolution

Run-time overload resolution, as needed in languages such as Haskell, is very similar to dynamic binding in object oriented languages. The central idea is to have "classes"

[3] We discuss continuations in Section 6.3.

of types on which overloaded functions (or methods) are defined. The main difference between this model and a general inheritance model is the restriction to a specific set of operators for which overloading is defined; fully general user class definitions may not be made although user datatype extensions may be added to existing class definitions.

In [44], Jones discusses using partial evaluation and specialization to eliminate the need for a run-time type *dictionary* in Haskell. Haskell implementations resolve run-time overloading by passing an additional parameter to all overloaded functions. The additional parameter is the dictionary which is equivalent to a method dispatch table in object oriented languages.

The use of dictionaries has many of the same properties as object oriented dynamic dispatch, but in particular, not being able to resolve the method being invoked means that most common program analysis techniques do not work very well. Haskell does allow the programmer to insert type information explicitly. This type information can then be used to remove the need for the run-time dictionary search; this is typically not possible in a more general inheritance framework.

The work by Jones is an interesting counterpoint to the object oriented work by Dean et al. Dean takes a more "pragmatic" approach to evaluating the profitability of particular specializations, while Jones has a more clean specialization algorithm. In both cases, the systems must deal with the potential for exponential code expansion due to unproductive specializations. In Dean et al this is done by making estimates of counts in a call graph while in Jones code expansion is controlled by a set of constraints on instances of calls in the code. As one example, Jones does not duplicate specializations through the use of memoization (Section 2.5). Each approach has drawbacks — Dean's system is less elegant and harder to implement than the Jones system but seems to yield better overall results. It is likely that some combination of Dean's model for the frequency of calls and the Jones model for the overall system might be an effective approach.

Run-time Code Generation

Although traditional compilers perform all code generation at compile-time, there has been considerable investigation into run-time code generation. Of these, the most notable is the Self [17].

One of the main issues during run-time code generation is when to spend time performing optimizations, or in other words, determining when the reduction in execution time is likely to be larger than the time spent on optimization. Leone and Lee [56] have investigated applying partial evaluation analysis techniques to this problem. Their basic approach is to introduce *late* and *early* annotations and to use these annotations to determine the code that is statically compiled (*early*) and the code that is dynamically compiled (*late*). Early code is compiled into code that performs any early operations while late code is compiled into code that *generates* the run-time code.

Leone and Lee compare this type of analysis with traditional binding time analysis techniques such as those used by Jones et al [48] and Consel [22]. The observation is that regular binding time analysis is more constrained in that there is an external division between static and dynamic annotations (corresponding to early and late annotations) while in run-time code generation, all of the static data is in fact available. The object of run-time specialization is not to take advantage of as much static information as possible, but rather to take advantage of the subset of static data that can lead to efficient code production.

The FABIUS system built by Leone and Lee performs an interesting form of code inlining; in fact, their rule is similar in flavour to a rule that we present in our on-line algorithm. In FABIUS, all loops are represented as tail-recursive functions. The inlining rule states that functions are only inlined if a *late* formal parameter does not appear in a branch of a conditional controlled by a late-stage value. In some senses, this can interpreted as saying that partial evaluation can only safely continue in the absence of dynamic conditionals. A similar statement will form part of the termination criteria for the on-line algorithm presented in Chapter 3.

2.6.2 Traditional Language Compilation

C program analysis

Andersen [5] [7] has investigated a very different domain – specialization of C programs. C is in many ways an extremely difficult language on which to apply partial evaluation techniques. C is highly imperative; nearly all operations in the language return values that can be assigned. Pointers are generally used with "wild abandon"

by C programmers and are often used in concert with run-time memory allocation. Coercions occur at many levels and alias relationships are very common. Finally, the exact semantics of many operations in C is dependent on the actual implementation making it extremely difficult to perform any substantial transformation and guarantee that the resulting program has the same behaviour over all run-time input as the original program.

Andersen followed an interesting approach in his work. Rather than directly interpreting the program source, his system creates *generating extensions* for the original source code. Generating extensions are not his innovation (see his thesis [7] for related work) but his particular application of the idea works well for C. A generating extension does not actually incorporate any static data into a new program, rather it is a program which given some static data actually generates the specialized program. Andersen's main motivations for following the generating extensions approach are, first, that extensions allow one to process the semantic information once as a separate issue from the specialization, and second, extensions defer the generation of new code and can be incorporated into an execution framework so that the generating extension and the final program execute under the same implementation-dependent semantics.

Andersen's work is an off-line approach in that he has separate phases to build the generating extension and to build the final specialization. He does not perform an automatic BTA on the C source but assumes the existence of binding time annotations. Although Andersen's use of generating extensions makes substantial progress towards performing reasonable transformations in an imperative environment, we feel that in order to have a fully automatic system that can perform non-trivial transformations, it will be necessary to bind the analysis and specialization phases more closely to the partial evaluation process.

Meyer [61] has also investigated imperative language specialization. His approach was between an on-line and off-line algorithm; he relies on initial annotations supplied by the programmer but then allows the annotations to change during the evaluation process. He does not directly address the relevance of on-line approaches but it seems that his approach could easily be subsumed by on-line approaches.

Fortran analysis

Fortran, although being an imperative language, is in many ways a much more "friendly" language for partial evaluation than is C. One of the groups investigating Fortran is Baier et al [9]. Their approach is fairly simplistic; they apply an offline, monovariant BTA to Fortran programs and then blindly specialize the resulting annotated programs.

Although the approach is not terribly sophisticated, the results are encouraging. On a number of common Fortran applications (FFT, cubic splines interpolation, and an n-body particle attraction problem), they achieved run-time decreases of 20–70%. Their observation was that many Fortran programs have large sections of code that are relatively independent of the dynamic data sets and were thus easily specialized. The specialization was primarily in the form of loop unrolling and they did not compare their resulting code to a compiler that performed aggressive loop unrolling. Their code sizes reflect this basic property of their algorithm — although they experienced code reduction of 50% on one small case, more typically specialization expanded the code by a factor of 10 to 100 on larger programs.

There are interesting questions that this work raises; the nature of the relationship between this type of approach and highly aggressive vectorizing compilers is unclear. It may be possible to use some of the unrolling analysis techniques used in vectorizing compilers to reduce the code expansion while retaining most of the speed improvement. Alternatively, it may be reasonable to attempt to regularize optimizations in the high-performance Fortran community be casting their approaches as instances of partial evaluation problems. Frameworks such as we propose could be a starting point for such a dialogue.

Incrementalization

Many programs calculate information redundantly as a result of particular methods for calculation. A classic case is the naive recursive definition of the Fibonacci numbers; using the naive algorithm, exponential time is required versus a reasonable intuitive linear time algorithm and a somewhat less obvious log-time algorithm. In [57], Liu, Stoller, and Teitelbaum present a method for automatically discovering inductive relationships in programs and then transforming the code into an incremental version in order to take advantage of the existing inductive relationships.

The presented approach is similar to naive partial evaluation with aggressive memoization and residual production. The only equality reasoning performed by Liu et al is based on symbolic term equality; there is no obvious reason why their approach could have stronger equality reasoning integrated into it. There has been substantial work performed in this area; we defer to Liu's paper for references to related work.

A similar approach that should be mentioned is the work done independently by Lawall [55] and Fegaras, Sheard, and Zhou [33]. In each case, the basic approach was to create systems that automatically reason about inductive structures. In Lawall's case, the actual transformations are then performed by hand, while in Fegaras et al, the transformations can automatically take place. This approach allows for non-trivial inductive reasoning and rewriting; Fegaras et al use an approach termed *catamorphisms* to describe types of inductive relationships for which automatic transformations are viable.

Other Types of Analysis

Several groups have investigate performing data flow analysis through partial evaluation [23][47] [81]. Partial evaluation naturally performs data flow estimates in order to calculate binding times in the off-line case, or as part of the interpretation in the on-line case. In either case, providing separate data flow analysis information does not require a substantial change in approach. As two specific examples, Vasell [81] uses an off-line approach in which the residuals generated by the "specializer" are in fact the data flow graphs of interest. The on-line Fuse [70] evaluator manipulates similar graphs as part of its internal analysis when representing "use" relationships.

Malmkjær, Heintze, and Danvy [58] perform partial evaluation on the LAMBDA intermediate form used in earlier versions of the SML/NJ compiler. LAMBDA is a continuations based (nearly) untyped intermediate form. The analysis performed by Malmkjær et al uses a simple set-based estimation to perform binding time analysis, control flow analysis, and data flow analysis. The set approximation approach adopted is a very conservative approximation; as one example, the analysis ignores all dependencies between variables. This causes substantial information loss if there are structural reorganizations, loop dependencies, etc. Although the approach proposed in this thesis has an aspect of set based analysis, set-based analysis seems to be much more valuable in an on-line environment where dependencies can be inter-

preted rather than ignored.

2.6.3 Other Applications

Ray Tracing

Ray tracing is in a sense the "first" application of partial evaluation. In 1986, Mogensen [65] proposed the use of partial evaluation for improving the performance of ray tracing algorithms. The most recent application of partial evaluation to ray tracing is work by Andersen [8] in 1995.

Ray tracing is a nearly optimal application for partial evaluation. Ray tracing is computationally expensive, there is a large static component (the scene) and there is a large interpretive overhead for dealing with the static component. The primary parameters in ray tracing are a set of objects, a set of light sources, an eye position and a window onto the scene. The window is a set of pixels that represent the scene at some given resolution with respect to the objects, light sources, and eye position.

Andersen made a fairly careful comparison with an efficient ray-tracing algorithm compiled under both *gcc* and a native platform (HP) compiler on an HP 9000/735. Specialized versions of the ray-tracing algorithm were built in order to take advantage of static knowledge regarding combinations of the three aspects in the algorithm. The specialized algorithms performed well in comparison to the original (optimized) code, ranging from a 20 to 70% reduction in computation time. The cost for the decreased computation time was an increase in code size by a factor of 1.1 to 10. Again, it is unclear whether this size/speed tradeoff is in fact close to "optimal"; it would be valuable to have a graduated specialization algorithm and attempt to characterize the point at which further unrolling is useless or even counter-productive.

Real-time Systems

Real-time systems can be partitioned into two broad classifications: *soft real-time* systems and *hard real-time* systems. Soft real-time systems are systems that have time constraints but where moderate violation of the constraints is not a critical problem. Examples of soft real-time systems include order display in a fast-food restaurant or frame update in a video game. In hard real-time systems, violating

time constraints can lead to catastrophic events. For example, missing a constraint in an automated production environment could lead to defective products or injury. Similarly, failure to meet constraints in an aircraft flight control system, particularly in a high-performance jet, may cause a crash.

In [68], Nirkhe and Pugh investigate the application of partial evaluation to a code for hard real-time systems. Perhaps surprisingly, this is an excellent application for partial evaluation. Typically, hard real-time systems disallow all computation paths that have unknown lengths implying that recursion, non-constant bound loops, and other non-constant cost operations are disallowed. Nirkhe and Pugh apply partial evaluation to transform programs that contain such features into systems that meet constant time operation constraints. Their contention is that by performing such transformations automatically, programmers can develop code at a higher level while maintaining the same hard guarantees.

The model chosen by Nirkhe and Pugh is very restricted. Part of this is due to the nature of the problem domain, but some of this is also their willingness to give up some expressiveness in order to have well-understood residual programs. For example, the store model adopted by this work splits the store into a purely compile-time component and a purely run-time component.

Nirkhe and Pugh use an off-line model. Their primary motivation for this choice is that having a separate BTA allows user interventions in the annotation process which in turn leads to tighter control over the characteristics of the final residual. In addition, they felt that handling global values within an on-line evaluation was problematic and that on-line approaches tend to over-specialize. Both of these concerns are addressed explicitly to some extent in this thesis (Section 6.3) and the general improvements in on-line approaches make these issues comparably difficult in automatic systems using both on-line and off-line techniques. The authors of this thesis do agree, however, that off-line systems do permit finer user control over annotations than current on-line processing and that this issue alone is sufficient to justify using only off-line approaches for hard real-time systems. There has not been any direct research into methods for allowing user intervention in the on-line annotation process. Although it would certainly be possible to allow user annotations to be introduced on an a priori basis, on-line evaluation is more interpretive in nature than off-line evaluation and as such, it would be more difficult to reason about the consequences of introducing particular annotations.

Deductive Database Query Optimization

Deductive databases are composed of a normal relational *extensional* database and a small *intensional* database consisting of a set of Horn clauses that define relations between tuples. In such systems, there are two aspects to evaluating a given query: evaluating the Horn clauses in the intensional database and performing the relevant queries on the relational database. When evaluating a query, the overall system can choose either to query the relational database and then interpret the Horn clauses on a tuple-by-tuple basis or it may choose to "compile" the Horn clauses into a series of relational database queries.

In [74], Sakama and Itoh report on the application of a simple partial evaluation model to deductive databases. Their basic approach is to first perform a partial evaluation of the Horn clauses and then to compile the remaining Horn clauses into relational queries using the normal method. Their partial evaluation consists primarily of unfolding Horn clauses until only recursive relationships or extensional queries remain. The method chosen by Sakama and Itoh ignores any binding time analysis and does as much unfolding as possible. The resulting system realized query execution improvements of 20 to 40% on large queries, but if the partial evaluation time is included, the improvement is nearly negligible. The system that they propose seems to consist solely of unfolding; there is little in the way of real specialization.

Related to this work is the larger body of work in applying abstract interpretation and partial evaluation to Prolog. In particular, there is a relatively early (1987) book [2] dealing with abstract interpretation techniques for declarative languages. More recent work in this area includes [13] [51][52][67] [69]. Another related topic is applying partial evaluation to solving systems of constraints [36] [76]. The basic observation in this work is that constraints have a "declarative" component and can be manipulated into a new system of partially solved constraints by applying Prolog-style rewritings.

Specification Verification

Sridhar and Vemuri [77] use partial evaluation for a rather different type of problem — verification of temporal specifications in hardware. This work defines a model for expressing hardware temporal constraints at the register transfer level. The language they define accepts trace behaviour and determines if the given traces conflict.

Partial evaluation is used to allow for partially unknown behaviour in traces which in turn allows classes of specifications to be validated simultaneously. Sridhar and Vemuri do not present the details of their approach, but their basic language is tightly constrained due to the nature of the hardware and thus seems as though it would be amenable to partial evaluation.

Chapter 3

Generalized On-line Partial Evaluation

On-line partial evaluation techniques do not use a distinct binding time analysis (BTA) preprocessing phase. As the specialization phase progresses, the partial evaluator decides whether it will treat each function call or variable as dynamic or static. This decision is only in effect for the current specialization decision; each specialization decision requires the partial evaluator to evaluate the status of each value involved in the specialization. This approach allows the specializer to change the status of any call at any point in the process.

In some ways, on-line techniques tend to be more complex than comparable off-line techniques. Off-line methods generally have a more modular aspect – there is a clear separation between the BTA and specialization phases. Off-line methods also have some advantages for self-application [46] as well as in allowing for user intervention in the annotations [26]. However, as discussed in earlier sections, on-line techniques have advantages in dealing with imperative features and in generalizing abstract values.

Our primary interest is in making use of partial evaluation techniques for optimization of traditional programs. Optimizations will occur during the intermediate phases of compilation. This criterion strongly influences our decision to use on-line methods. Self-application (see the discussion on the Futamura projections in Section 2.2) is not an issue and it is unlikely that we would want the user to have direct

influence over annotations, although we may want to allow indirect user influence through the weighting of various optimization tradeoffs (i.e. size/speed).

Our approach has several key features: it incorporates uncertain knowledge, it promotes a consistent mechanism for modeling program behaviour, and it incorporates a consistent termination mechanism. The approach that we will introduce will incorporate very general domains for specialization. These domains will cause increased complexity in the termination criteria, but will allow a single method to address concerns about imperative features and normal polyvariant specialization.

3.1 Domains for On-Line PE

In Section 2.4.2 we noted that traditional off-line methods do not adequately model imperative language features. The primary reason for the weaker model is in the approach to safety and termination. Termination and safety proofs for off-line systems rely on having a fixed-height lattice representing knowledge about the system. Intuitively, this restriction guarantees that the systems will always make progress towards a solution (fixed-point) that is at most some fixed distance away. Unfortunately, finite lattice structures cannot adequately model uncertain or partial knowledge in a system.

Consider a statement such as the following:

```
if x = 5 then
   y := 7
else
   y := 5;
```

In traditional systems, if x is dynamic we cannot model the value of y after this statement, other than to say that y is an integer (which we may already know if the language provides that information through the type system). Intuitively, however, we realize that treating y as dynamic does not adequately reflect what we know about y, namely that after the statement we know that y has either the value 5 or the value 7. We may not know which value y contains, but we do know that there are a finite number of options. We could then use this information to make further specializations. For example, assume the following statement came next:

```
if y < 10 then
   z := 7;
```

With a traditional approach, y would be considered as dynamic and no specialization could occur. In our system we would realize that this code fragment has the same behaviour for *all possible* values of y or in other words, that z will definitely be assigned the value 7. In this case the entire statement might disappear since any subsequent use of z would be replaced by the static value of z, that is, 7. This general approach will also be used to model structures as will be discussed in Section 3.1.2.

There are some difficulties with this approach. Due to the nature of the resulting domains, a lattice model of the domain is no longer of finite height. If we allow sets of values into our model, we will often encounter infinite sets of values in recursive code when the termination condition for the recursive code is dynamic. In a traditional approach this does not pose much of a problem since the BTA will treat the problematic variable as dynamic and the specializer will not have to deal with it. Our approach models the growth of a set of values and determines when to "give up" and treat the variable as dynamic. Due to termination concerns, our approximation to these sets of values will have to be conservative, but will be able to model partial knowledge more completely than existing systems.

3.1.1 Domain Approximations

The easiest technique for dealing with sets of values in domains is to use a completely ad hoc technique. For example, simply using sets of values as an approximation and inserting new values into the set as they are encountered is a viable approach. In order to determine when to convert the value to a dynamic status, the cardinality of the set could be used. In other words, as long as the set has a cardinality of less than some "trigger" value, we continue to add elements to the set. If the set cardinality surpasses the trigger limit, we begin treating the variable as fully dynamic. Under a lattice model, cardinality is actually quite clean; the lattice has one level for each possible cardinality of sets, with $\top$ being above the level representing sets with the highest permitted cardinality. Figure 3.1.1 represents a lattice for the subset relationship with sets of cardinality less than or equal to 4. Each level consists of an infinite number of sets of the given cardinality. Each of these sets is a subset to an infinite number of sets at the next level up the lattice. Sets of cardinality four are all members of $\top$, which represents the set of all integers.

Although cardinality can be used, there are several problems with this method.

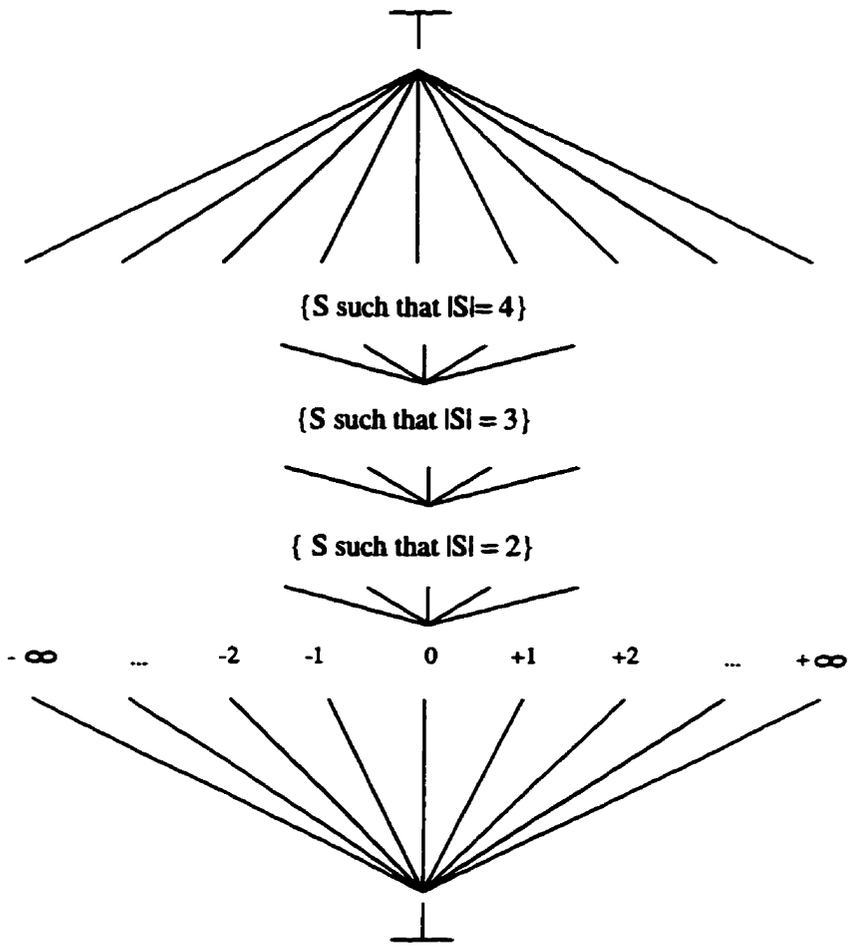


Figure 3.1.1: Restricted Subset Lattice

First of all (and most importantly), we may perform a great deal of work before deciding to treat the variable as dynamic; the computational expense would discourage practical use of the techniques. Second, one of the goals of this work is to move beyond informal techniques. Although this would be a more powerful model than a simple lattice model, we would prefer to have a more consistent approach.

The basic approach that we adopt is to create a general method that is roughly based on work done by Cousot and Cousot [27], and Bourndoncle [14]. We need to have a finite model of infinite domains, but we also need to have a computationally inexpensive process for making the estimate. When using sets of values, when an element is added to a set there is no analysis of the set itself. This leads to the problems noted above. In [14], Bourndoncle presents a method for approximating the behaviour of functions by using a pair of intervals. The first interval gives the range of the input arguments and the second gives the interval of the output of the function. For example the interval pair <[1, 5], [10, 50]> would represent a function which, when given values in the range 1 to 5, produces values in the range 10 to 50.

Bourdoncle generates these intervals by applying a *widening* operation, ∇. The definition of ∇ over integers (∇_I) [14] is as follows:

$$\begin{aligned} &\bot \nabla_I x = x \nabla_I \bot = x \\ &[a_1, b_1] \nabla_I [a_2, b_2] = [\quad \text{if } a_2 < a_1 \text{ then } \infty^- \text{ else } a_1, \\ &\qquad\qquad\qquad\qquad \text{if } b_2 > b_1 \text{ then } \infty^+ \text{ else } b_1 \ ]. \end{aligned}$$

This operator is very conservative – if you attempt to extend a range in either direction, the range is extended to infinity. Essentially this models a function with a "base case" and a general case; the base case will be the start of the interval and the interval will extend to infinity. This type of estimate is not usually very informative due to its very conservative nature. Bourndoncle does introduce more precise widening operators, but does not give any formal framework for deciding which operator to use for a given widening.

The partitioning work done by Bourndoncle estimates the behaviour of programs by using abstract control points at which intervals are calculated. The abstract control points partition the (often infinite) set of program control points (the set of run-time program states) into a finite set which are used to determine intervals.

Interval pairs will be used in our approach to estimate the behaviour of functions and to create equivalence classes of functions (which in turn determines termina-

tion). The fundamental operation in this approach is to *widen* domains using the (∇) operator. The widening operator is a conservative over-estimator for domains; it can be seen as an imprecise join.

Bourndoncle's approximations are built by successively widening the input specification by the next approximation to the program's meaning. The program's meaning is approximated by a safe abstract meaning function $\Phi^{\#}$, which is defined individually for each program.

Bourndoncle's approach is similar to what we will propose – the primary difference is that Bourndoncle does not discuss unknown values as part of the input specifications, nor how to automatically infer $\Phi^{\#}$. Our approach must be able to deal with both issues. In addition, since these estimates are performed in order to permit termination decisions to be made, we must distinguish several estimates for the same function. For example, consider the following:

```
(define f
    (lambda (x)
        (+ x 10)
))

(f 4)
(f z)
```

where `z` is unknown. We must not include both of the calls to `f` when constructing the domain estimate for `f`. If both estimates were included in the domain, we would lose all information about the static value in the first call. In our work, each of the calls causes a distinct polyvariant specialization; there is no interaction between the two specializations. In general, the only time at which a call effects the domain estimate for another call to the same function is when the second call occurs within the first, i.e. when either direct or indirect recursion occurs.

3.1.2 Issues for Structured Domains

A *structured type* is a composition of basic types using type constructors. Simple examples include arrays, lists, records, and trees. Elements of structured types may be composed of many simpler elements. Structures may be approached in one of two ways for the purposes of binding time analysis: the entire structure may have

a single annotation, or each element within the structure may have a separate annotation. Using only a single annotation significantly restricts the accuracy of the partial evaluator. Single annotations for structures correspond to monovariant BTA for functions; the annotation for the structure must be the least upper bound of the element annotations. With this approach structures are only considered to be static if *all* elements of the structure are static.

Consider the following example:

```
(define head
    (lambda (x)
        (car x)))

(head (list '1 z))
```

where z is dynamic. With a single annotation, the list resulting from the cons will be dynamic and the entire call to head will remain in the residual. The annotations would be as follows:

```
(define head
    (lambda (x)
        (car x_D)))

(head_D (list '1 z)_D)
```

Using separate annotations for elements within structures would result in the following annotations:

```
(define head
    (lambda (x)
        (car  x_(s,D))))

(head_s (list '1_s z_D))
```

Within the function head, the variable x has two annotations, each of which refers to the corresponding element in the structure. The residual in this situation is much more pleasing; car only depends on the first element of the structure so we can completely evaluate the function and remove the call to head from the residual.

Structural decomposition of a domain may be approached in several ways. As alluded to in Section 2.4.2, one method is to factor structured types into simpler components. One must however, take great care not to change the semantics of the source language. Since languages have different semantics for structural and base types, using this as a general approach seems problematic at best.

Regular Expression Notation

A more interesting avenue for further exploration involves the actual domain representations for structures. In [40] and [39] Hendren introduces a regular expression notation for describing structures. For binary trees, the regular expressions are composed of a series of the following symbols:

- *S*: no edge (*S*ame node)
- *L*: a *L*eft edge
- *R*: a *R*ight edge
- *D*: a *D*own edge (either a right or left edge)

Each of these symbols may be repeated or may have a superscript denoting the number of instances of the symbol. Thus LL or L^2 would both represent two left links. A superscript of "+" indicates one or more links, while a "?" following a term indicates zero or one occurrences of the term.

Given a path expression such as R^2LD+ for the path from the root of a binary tree to a node, `c`, we would have the tree shown in Figure 3.1.2.

Hendren develops a calculus for manipulating expressions and is able to handle possible paths as well as certain paths. For example, consider the following imperative code:

```
if x < y then
    a.left := c
else
    a.right := c;
c.left := d;
```

Assuming that the truth of the conditional is unknown, the path from `a` to `c` after this code fragment will be D and the path from `a` to `d` will be DL. In [40], Hendren gives the following example (with one variable renamed for clarity):

```
c := h;
while c.left <> nil do
    c := c.left;
```

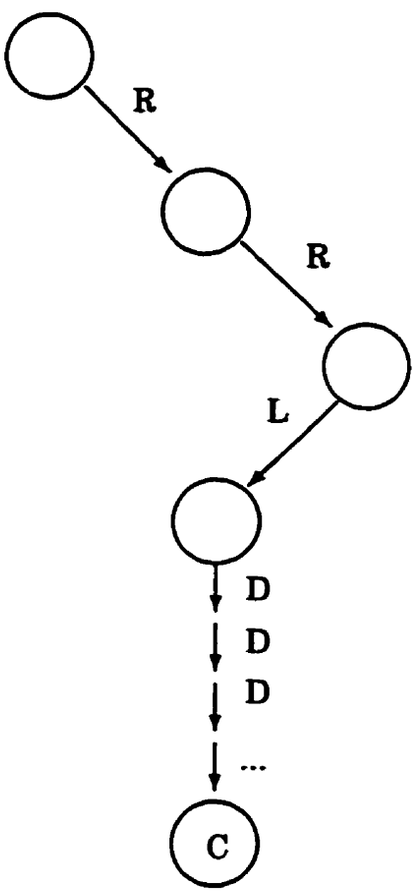


Figure 3.1.2: A tree for R^2LD+

The path from h to c is approximated iteratively until a fixed point is reached; the approximations are S, $\{S, L\}$, and $\{S, L^+\}$ which is the fixed point. This means that either h and c are the same node, or there is a series of one or more left links between h and c.

Hendren's approach to structured domains is essentially the same as Bourndoncle's approach to non-structured domains. Consider the following code:

```
c := 0;
while c <> z do
    c := c + 1;
```

where z is unknown. Approximating this domain using Bourndoncle's approach results in an approximation of [0..∞] for c. This is essentially the same result as Hendren's, as [0..∞] can be understood as the set of possible "distances" from 0 to c which is what the S, L^+ expresses in the structured domain. Hendren's approximations are more accurate in some cases; consider the previous example where Hendren's approach captures uncertain knowledge about the direction of the link (we couldn't tell whether the link was R or L, but could still express the link as D). Bourndoncle's approach, using the simple widening operator, would extend the domain to infinity, losing some of the information.

Hendren's calculus can be understood as a set of widening operators which are somewhat more precise than Bourndoncle's simple widening operator. Unfortunately, the cost for the increased precision is incorporating knowledge about the data structure into the model. In order to model trees, Hendren has specific abstract values for *left* and *right* links. When the model is extended to a simple DAG, a third type of link, M (middle), needs to be introduced.

In general, we will not know in advance what the data structure will look like, and thus we will not be able to generate estimates with the same level of accuracy as Hendren's approximations. On the other hand, our approach will not need the specific knowledge required in Hendren's approach. Finally, Hendren's goals are to characterize the paths between nodes; we are concerned with the values on those paths in addition to the path descriptions.

3.2 Improving Domain Approximations

There are two types of widenings that we will want to perform: a *precise* widening and a *relaxed* or general widening. Consider the following imperative example:

```
if x < y then
   z := 5
else
   z := 7;
```

If the conditional is static, it is easy to see how to generate exact knowledge about z. If, however, the conditional is dynamic, we would like to be able to generate the precise information that z contains either a 5 or a 7. Using an imprecise widening operator we would extend the domain of z from 5 to ∞, which is not a very reasonable estimate even though it is "correct".

Consider also the following two code fragments:

```
z := 0;
while x < y do
   z := z + 1;
   x := x + 1;
```

```
z := 0;
while x < y do
   z := 6;
   x := x + 1;
```

In both code fragments, if x and y are both static then we can completely evaluate the loop and have a single static value remain for z. If, however, the conditional is dynamic, we should not treat the code fragments in the same way. Consider the first fragment. If the conditional is dynamic, the best estimate that we can make for z is the interval $[0..\infty]$. In the second code fragment, however, the best estimate for z is the pair of (singleton) intervals $[0..0], [6..6]$. The reason that there is a difference in the best result is that in the second case we have a constant result and in the first case we have a computed result which depends on a dynamic value.

Although this example only deals with integers, we can construct examples that demonstrate similar concerns in other domains (characters, boolean values, lists, etc). Our basic approach to dealing with the problem of using only the basic widening operator is to define two widening operators, each of which will be used in the appropriate situations.

In the remainder of the chapter we will define the on-line partial evaluation algorithm. The development will occur in three steps: first we define the properties that abstract domains must satisfy, we then define properties of the widening operators, and finally, we define the partial evaluation algorithm by appealing to these properties.

3.3 Domains and Widening Operators

When a non-abstract (or *concrete*) interpreter is defined for a given language, the interpreter will incorporate knowledge of various types into its operation. Examples of such types include integer values, booleans, characters, lists, etc. As mentioned in Section 1.4, in order to perform an abstract interpretation for a given language, we must define abstract domains that correspond to each type that a non-abstract interpreter would use. We will refer to the set of values represented by a type in the standard semantics as the *natural concrete domain*.

Programming languages normally define primitive operations over the natural concrete domains; we must define corresponding abstract operators over the abstract domains. We must also have some (minimal) guarantees about the behaviour of the abstract domains in order to be able to build a consistent abstract interpretation. Finally, we must have a method for transforming a concrete value into an abstract value, and for transforming an abstract value into a concrete value. Note that for a given natural concrete domain it may not always be possible to transform an abstract value into a particular natural concrete value; for example, consider our *negative/zero/positive* example from the introduction. It is simple to convert any concrete natural number into the *negative/zero/positive* lattice, but it is not possible to convert a *positive* or *negative* abstract value into a single natural number.

In the next section we define the properties that abstract domains must satisfy. These definitions will apply to *all* abstract domains. As will be seen in Section 4.2, the termination and correctness of the partial evaluation algorithm depend on only the general properties of each abstract domain; there is no dependence on any actual abstract domain. From a design perspective, this allows a clean distinction to be made between the partial evaluation algorithm and the actual abstract domains used in an implementation. In addition, assuming that we have a proof of correctness for the system that depends on only the domain properties, we can then reduce a proof of

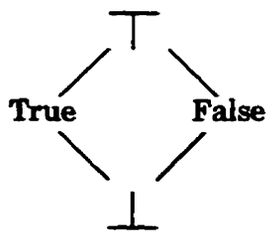


Figure 3.3.1: Boolean Concrete Domain

correctness for an entire system to a proof that a set of given actual abstract domains satisfies the given domain and operator requirements. This will be the approach we adopt in Chapter 5 when we present abstract domains for the integer and list domains.

Finally, it is also important to note at this point that the termination and correctness proofs are not related to the *accuracy* of the overall system. The accuracy of an implementation depends primarily on the abstract domains that are used in a particular implementation. If one wishes to have a more accurate interpreter, more accurate domains may be introduced; the only *requirement* is that the actual abstract domains satisfy the given constraints.

3.3.1 Domain Requirements

Within our system we will not use the natural concrete domains directly. Concrete values used by our system will be taken from the complete lattice formed by *lifting* and *topping* the corresponding natural concrete domain. Lifting simply introduces a $\perp$ element and topping introduces a $\top$ element. We will refer to the lifted and topped natural concrete domain as the *concrete domain*. This construction is important as it allows the interpretation algorithm to determine the accuracy of abstract values. As one example, Figure 3.3.1 shows the concrete domain lattice that corresponds to the "boolean" natural concrete domain.

In order to improve the accuracy of our results, we do not simply use least upper bounds on lattice values since least upper bounds can over-generalize abstract values. Rather, we want to have the interpreter decide when to make the conservative compromise between accuracy and termination. In order to meet this goal, values in

the abstract domains (or simply *domains*) are composed of a set of incomparable elements where each element is chosen from some lattice. As reviewed in Appendix A, lattice elements x and y are incomparable if $x \not\preccurlyeq y$ and $y \not\preccurlyeq x$. We denote the fact that x and y are incomparable as $x \| y$.

The widening operations are defined in terms of a modified definition of *down-sets*. The normal view of down-sets is discussed in Appendix A; we will briefly review the concept here as well. The basic idea of a down-set is that the down-set of a lattice element e, denoted $\downarrow e$, is the set of elements below (or equal to) that element within the lattice. We may also apply the idea of a down-set to a set; the down-set of a set of elements is simply the union of the down-sets of each element.

The abstract models we are interested in can be slightly constrained from the normal fully general lattices — we are interested in modeling information about natural concrete domains. Natural concrete domains (or normal types) are basically sets of elements. Although these sets may be ordered by various relational operations, they are not ordered in terms of "meaning". In other words, in any natural concrete domain, there do not exist distinct elements, x, y such that x subsumes the meaning of y. This means that every natural concrete domain is composed of elements which are incomparable to any other element in that natural concrete domain. This means that within the concrete domains, if $x \prec y$ then either $x = \bot$ or $y = \top$.

Normally a down-set for a lattice element includes *all* elements in the lattice that are below the given element. We modify this interpretation to include only the lattice *atoms* in the down-set. An *atom* in a lattice is a value x such that $\bot \prec x$ and if $y \neq \bot$ and $y \preccurlyeq x$ then $y = x$. Intuitively, the atoms are the values in the lattice that are immediately above $\bot$. In terms of the concrete domains, the atoms of the concrete domain are exactly the elements of the natural concrete domain. This in turn has a direct correspondence to what we want our lattices of abstract values to mean — we want the lattices to express information about some subset of the elements in the natural concrete domain.

For the rest of the presentation, we will use $\downarrow V$ to represent only the atoms below V. Given this definition of down-sets, it is straightforward to extend the normal lattice ordering relationship to sets of elements. Given sets of lattice elements, x and y, we will say that $x \sqsubseteq y$ if $\downarrow x \subseteq \downarrow y$. We will reuse the term "below" for $x \sqsubseteq y$; although this overloads the term with the basic lattice relational operator "$\preccurlyeq$", conceptually the two operators have similar semantics. Note that we use the term "below" to mean

"below or equal to"; when we intend a strict relationship, we will use the term "strictly below". We use the terms "above" and "strictly above" in a similar way. Finally, we define equality by saying that two values, x and y, are equal if $\downarrow x = \downarrow y$.

There are two additional comments that need to be made regarding this formal model. First, as discussed by Davey and Priestley [30], given an ordered set P, the set of all down-sets of P, represented as $\mathcal{O}(P)$, is a complete lattice under subset inclusion. In effect, we will be finding upper bounds and least upper bounds in $\mathcal{O}(P)$. We choose to ignore this aspect as we feel that simply talking about the down-sets themselves is a more intuitive approach while sacrificing no formal expressiveness. Note that this applies to our modified interpretation of down-sets as well; the sets are simply members of the powerset of P. The second comment is that these definitions are *behavioural* and not *operational*. For all but trivial base lattices, operationally manipulating down-sets is computationally prohibitive. In practice however, most "intuitive" abstract lattices lend themselves quite well to this behavioural description while retaining efficient computation characteristics. This issue is developed further in Chapter 5 when we present particular abstract domains.

3.3.2 The Widening Operators

All widening operations are performed on abstract domains; although we will talk about performing widening operations on values in *domains*, the reader should keep in mind that the domains will always be *abstract domains* that satisfy the requirements discussed in the previous section and in particular that any value in a domain is a set of incomparable elements.

The basic idea of any widening operator is to coalesce two pieces of abstract information; the nature of the resulting value depends on the actual (abstract) domain and the type of widening that is performed. There are two types of widening operators that our algorithm uses: a *precise* widening operator and a *relaxed* widening operator. A *precise* widening of two values (sets) of a domain results in a value which we expect to represent *only* those elements present in the two original values. A *relaxed* widening will result in a value which includes *at least* those elements in the two original values.

We will use ∇_P to denote a precise widening operator; $\nabla_P^{\mathcal{D}}$ will represent a particular precise widening operator over a domain $\mathcal{D}$. Normally, we will not explicitly

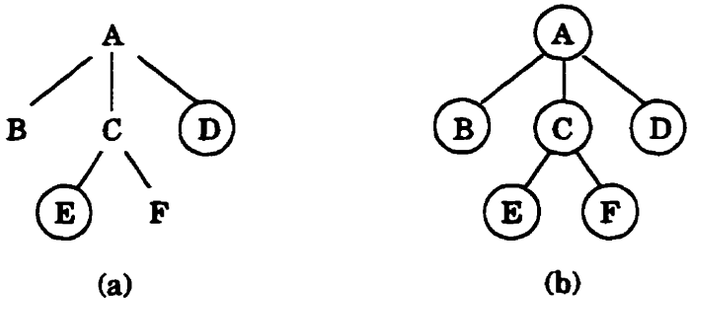


Figure 3.3.2: (a) $E \nabla_P D$ and (b) $E \vee D$

denote the domain of a widening operator. The conditions for precise widening are summarized in Definition 3.1.

Defn 3.1 (Precise Widening) *If V_1 and V_2 are values in some domain, then ∇_P is an operator such that*

$$V_1 = V_1 \nabla_R V_1$$

$$V = V_1 \nabla_P V_2 \Longrightarrow \begin{cases} \downarrow V = (\downarrow V_1) \bigcup (\downarrow V_2) & (i) \\ \forall x, y \in V : x \neq y \Longrightarrow x \| y & (ii) \end{cases}$$

On occasion it is useful to consider the result of widening several values; we will use the notation $\nabla_P (V_1, V_2, \ldots)$ to mean $(\ldots((V_1 \nabla_P V_2) \nabla_P V_2) \nabla_P \ldots)$.

As noted earlier, the proposed model generalizes other models and formalizes the precision that we want in our model. In general, a simpler approach based strictly on least upper bounds in the underlying lattice rather than down-sets, can over generalize results. For example, Figure 3.3.2 shows the difference in the accuracy between joining two elements and taking the union of their down-sets. In each case, the elements covered by the result are circled. Although this is a somewhat contrived example, similar behaviour is manifested in interval lattices and other relatively intuitive lattices.

There are two fundamental differences between precise and relaxed widening. The first difference is that relaxed widening is less restrictive than precise widening

about the accuracy of resulting values. The second difference is that relaxed widening operators must guarantee that widenings cannot occur indefinitely without converging to some stable value.

We will use ∇_R to denote a relaxed widening operator; $\nabla_R^{\mathcal{D}}$ will represent a particular relaxed widening operator over a domain $\mathcal{D}$. The conditions for relaxed widening are summarized in Definition 3.2.

Defn 3.2 (Relaxed Widening) *If V_1, V_2, W_1, and W_2 are values in some domain with $W_1 \sqsubseteq V_1$ and $W_2 \sqsubseteq V_2$, then ∇_R is an operator such that*

$$V_1 = V_1 \nabla_R V_1$$

$$V = V_1 \nabla_R V_2 \Longrightarrow \begin{cases} \downarrow V \supseteq (\downarrow V_1) \bigcup (\downarrow V_2) & \textit{(i)} \\ \forall x, y \in V : x \neq y \Longrightarrow x \| y & \textit{(ii)} \end{cases}$$

$$W_1 \nabla_R W_2 \sqsubseteq V_1 \nabla_R V_2$$

and

for any function f and value x_0, there exists a k such that $f(x_k) \sqsubseteq x_k$ where $x_i = x_{i-1} \nabla_R f(x_{i-1})$ for $i > 0$.

Note that in the definition for ∇_R, we only require that the resulting abstract value include at least the values in the two original abstract values. This implies that using traditional least upper bound approximations would be acceptable for relaxed widening operations.

This approach to modeling values has two major advantages in comparison to the finite height lattice model adopted by other systems. First, this model allows us to differentiate between generalizing values to capture program information and generalizing values for termination purposes. The former can be done exactly while the latter must be done in a more conservative manner in order to guarantee termination. Second, incorporating the convergence requirements with value operations allows the operators to take advantage of the values that are being manipulated. In effect this allows the operators to create a finite height projection of an infinite height lattice during the evaluation. This allows interaction between the program and the

actual set of abstract values rather than determining the entire abstract model before any evaluation.

3.3.3 Other Requirements

Abstraction and Concretization

For each concrete domain and corresponding abstract domain, there must be an *abstraction* function and a *concretization* function. Following Jones et al [46], we will represent an abstraction function as α and a concretization function as γ. Given a value, v, in a concrete domain, $\alpha(v)$ is the corresponding abstract value for v. Given an abstract value, v', in some abstract domain, $\gamma(v')$ is the corresponding concrete value for v' in the (lifted and topped) concrete domain. As in Jones et al, we require that the abstraction and concretization functions be *monotonic*. A function, f, is monotonic if $a \sqsubseteq b$ implies that $f(a) \sqsubseteq f(b)$.

Jones et al requires that for every abstract value, transforming the abstract value into the concrete domain and then back into the abstract domain is an identity operation. More formally, for a given abstract domain it is required that $\forall s \in \{\text{Abstract Domain}\} : \alpha(\gamma(s)) = s$. We weaken this requirement to say that converting an abstract value into the concrete domain and then back to the abstract domain generates a value which is above the original value. More formally, we require only that $\forall s \in \{\text{Abstract Domain}\} : s \sqsubseteq \alpha(\gamma(s))$. Finally, we follow Jones et al in requiring that for every concrete value, transforming the concrete value into the abstract domain and then back into the concrete domain yields a value that is above the original value. Formally, this is stated as $\forall s \in \{\text{Concrete Domain}\} : s \sqsubseteq \gamma(\alpha(s))$.

We require that abstracting a concrete value s produces a minimal, non-bottom abstract value that can be used to represent s. Formally, this means that if $\alpha(v) = a$ such that $\gamma(a) \sqsupseteq v$ then $a \neq \bot$ and there does not exist $a' \sqsubset a$ where $\gamma(a') \sqsupseteq v$. We further extend the meaning of the abstraction function, α, such that if op is a primitive operation in the interpreter, then $\alpha(\text{op})$ is the abstract operation corresponding to op. Given an n-ary primitive function, op, the requirement for $\alpha(\text{op})$ is as follows: given a set of values $v_1, v_2, \ldots, v_n$ and corresponding abstract values $v_1^\alpha, v_2^\alpha, \ldots, v_n^\alpha$ such that each $v_i^\alpha \sqsupseteq \alpha(v_i)$ then $(\alpha(\text{op})\ v_1^\alpha, v_2^\alpha, \ldots, v_n^\alpha) \sqsupseteq \alpha(\text{op}\ \ v_1, v_2, \ldots, v_n)$. In addition, we require that $\alpha(\text{op})$ only produces $\bot$ if one of its arguments is $\bot$ or if the operator is not total and the result of $(\text{op}\ \ v_1, v_2, \ldots, v_n)$ is not defined. In other words,

$(\alpha(\mathrm{op})\ v_1^\alpha, v_2^\alpha, \ldots, v_n^\alpha) = \perp$ implies that either for some $1 \leq i \leq n, v_i^\alpha = \perp$, or that $(\mathrm{op}\ v_1, v_2, \ldots, v_n)$ is not defined.

Domain Splitting

We require that every domain provide a function, *Split*, that performs value "splitting" over the relational operations that are defined in the domain. For example, given a relational expression such as $x < y$ over an abstract domain that supports less-than comparisons, *Split* provides a pair of abstract values: the first abstract value is a subset of x that contains at least those values that satisfy the relation; the second abstract value is a subset of x that contains at least those values that do not satisfy the relation. Note that we don't require *Split* to be "accurate", only "safe" in the sense that each of the pair of resulting values is a superset of the set of values within x that satisfy or don't satisfy the given relation. *Split* could safely return a pair in which each value is the original value x. The splitting function will be used when we evaluate conditional expressions; it allows us to build "custom" environments for each branch in the conditional. The full definition of *Split* depends on the definition of the standard semantics; we will more carefully define *Split* in Section 3.5.3.

3.4 The Language and Standard Semantics

There are three important aspects to any on-line partial evaluation algorithm. First, the ability of the algorithm to retain static information directly determines the quality of results. Second, the algorithm must have some method for dealing with the issue of divergence. Finally, the algorithm must be sound, or equivalently, must produce correct answers. In order to simplify the presentation and to focus more clearly on the contributions, we use a very simple language for the interpreter. The language is a first-order, pure, functional language similar in form to Scheme [19] or Lisp [78]. Although it is possible to introduce simple approaches for dealing with higher-order functions, non-trivial approaches have not been investigated in any partial evaluation work; this is discussed further in Section 7.2.2.

We assume that there are a finite number of functions; each function, $\lambda \mathtt{x}.\mathtt{e}$, is identified by a unique identifier, $\lambda \mathtt{x}.\mathtt{e}_{id}$. When we give the semantic definition for

function application, we assume that the function identifier is replaced by the definition of the function. For the purposes of examples, we will use function names as the function identifiers. The basic BNF of the language is as follows:

$$\begin{aligned} \texttt{E} ::= &\texttt{(if E E E)} \mid \texttt{(}\lambda\texttt{x.e}_{id}\texttt{ E)} \mid \\ &\texttt{(op E*)} \mid \texttt{const} \mid \texttt{ident} \end{aligned} \tag{3.1}$$

In general, an expression, e, may interact with the external world. For the simple language we are defining, we require that all such interactions occur through the initial identifier environment used in evaluating e; in other words, all "dynamic" or "run-time" information is provided to e by way of this initial environment. All free variables in e are assumed to use dynamic data and thus the initial identifier environment for $\mathcal{N}$ contains bindings to concrete values for all free variables in the expression e. This implies that we also assume that functions do not have free variables other than to dynamic input values. Finally, we restrict non-primitive functions to being single argument (monadic) functions. The restrictions regarding free variables and monadic non-primitive functions are not fundamental but substantially simplify the soundness statement and proof presented in Chapter 4.

We define the semantics for our language by giving an operational semantics labeled $\mathcal{N}$. The semantics are defined in terms of a source expression and an environment. $\mathcal{N}$ produces an expression representing the result of evaluating the expression. Symbolically we represent the general form as $\mathcal{N}[\![\texttt{e}]\!]\,\varrho = e'$. The environment ϱ contains a mapping for each identifier to a value for the identifier. Thus $\varrho(\text{id}) = \texttt{const}$ for some constant value `const`. $\mathcal{N}$ may not be defined for particular expressions. In particular, if a primitive is not total, $\mathcal{N}$ may be undefined.

Constants

The interpretation of a constant is simply the value of the constant.

$$\mathcal{N}[\![\texttt{const}]\!]\,\varrho = \texttt{const} \tag{3.2}$$

Identifiers

The interpretation of an identifier is simply the value bound to the identifier within the current environment.

$$\mathcal{N}[\![\texttt{ident}]\!]\,\varrho = \varrho(\texttt{ident}) \tag{3.3}$$

Conditions

The value of a conditional expression is the value of the appropriate branch of the expression. The branch is selected based on the result of evaluating the controlling condition. Allowing side-effects would involve having the evaluation of the condition return a modified environment that would then be used for the evaluation of the branches.

$$\mathcal{N}[\![\texttt{(if c } e_1\ e_2\texttt{)}]\!]\varrho = \tag{3.4}$$

$$\text{let } c' = \mathcal{N}[\![\texttt{c}]\!]\varrho \tag{1}$$

$$e' = \begin{cases} \mathcal{N}[\![e_1]\!]\varrho & \text{if } c' = \texttt{true} \\ \mathcal{N}[\![e_2]\!]\varrho & \text{if } c' = \texttt{false} \end{cases} \tag{2}$$

$$\text{in}$$

$$e' \tag{3}$$

$$\text{end}$$

Primitive operators

The evaluation of a primitive function requires the evaluation of the arguments and then the application of the primitive to the resulting argument values. We will not concern ourselves with a particular set of primitives but assume the existence of a "sufficiently rich" set. Allowing side-effects would simply involve having a modified environment returned from each evaluation and passing the modified environment to the next evaluation.

$$\mathcal{N}[\![(\mathrm{op}\ \ \mathrm{e}_1\,\mathrm{e}_2 \ldots \mathrm{e}_n)]\!]\varrho = \tag{3.5}$$

let

$$\mathrm{e}_i' = \mathcal{N}[\![\mathrm{e}_i]\!]\varrho \qquad \text{for all } 1 \le i \le n \tag{1}$$

in

$$\mathit{apply}(\mathrm{op},\ \mathrm{e}_1'\,\mathrm{e}_2' \ldots \mathrm{e}_n') \tag{2}$$

end

Function Application

A non-primitive function application involves little more than the evaluation of a primitive. First we evaluate the argument, then we create a modified environment containing a binding from the formal argument to the actual argument value, and finally, we evaluate the body of the function in the context of this new environment. As with primitive function application, it is straightforward to extend the rule to allow impure, polyadic functions.

$$\mathcal{N}[\![(\lambda \mathrm{x}.\mathrm{e}\ \ \mathrm{e}_1)]\!]\varrho = \tag{3.6}$$

let

$$e_1' = \mathcal{N}[\![\mathrm{e}_1]\!]\varrho \tag{1}$$

in

$$\mathcal{N}[\![\mathrm{e}]\!]\,\varrho[\mathrm{x} \mapsto e_1'] \tag{2}$$

end

3.5 The Online Algorithm

Symbolically, we will denote our partial evaluator as $\mathcal{P}$. The partial evaluator $\mathcal{P}$ takes a source expression, two environments, and a boolean flag and produces a pair containing an abstract value representing the result of the expression and a residual for the source expression. Symbolically we represent the general form as the following:

$$\mathcal{P}[\![\mathrm{e}]\!]\rho\,\delta\,\xi\,d \rightarrow < \mathrm{e}^{\alpha}, \mathrm{e}^{R} >$$

We will use the superscript α on variables to denote that they represent an abstract value and will use the superscript R to denote variables that represent residuals. The first environment, ρ, is an environment mapping each identifier to a pair containing the abstract value and current residual for the identifier. Thus $\rho(\mathrm{id}) =< \mathrm{id}^{\alpha}, \mathrm{id}^{R} >$. The second environment, δ, maps function identifiers to estimates of function arguments. As stated earlier, we assume that each function, f, can be identified by a unique identifier f_{id}. Given a function identifier, f_{id} and a function application $f(v)$, we then have $\delta(f_{id}) = v^{\alpha}$. The ξ environment maps function identifiers to estimates of function values. Given a function identifier, f_{id} and a function application $f(v)$, we will have $\delta(f_{id}) = f^{\alpha}_{id}$. The final parameter, d, is a boolean flag that represents whether the current evaluation path through the source contains a dynamic conditional statement.

Given an expression e, the initial evaluation of e will have a ρ environment binding all free variables in e to $\top$, δ and ξ environments binding all function identifiers to $\bot$, and will have d = false. Recall that the initial environment captures the "runtime" values for e; binding all free variables in e to $\top$ in the initial environment means that each free variable has an "unknown" value during evaluation with $\mathcal{P}$.

We will use the notation $\mathit{first}(t), \mathit{second}(t)$, etc. to represent element projection from a tuple.

3.5.1 Constants

The simplest case for $\mathcal{P}$ involves a constant expression. The value of a constant is simply the corresponding abstract value for that constant and the residual of a constant is the constant itself. Recalling that α is our abstraction function, in our

symbolic form the behaviour of $\mathcal{P}$ for constants is expressed as the following:

$$\mathcal{P}[\![\texttt{const}]\!]\rho\,\delta\,\xi\,d = < \alpha(\texttt{const}), \texttt{const} > \tag{3.7}$$

3.5.2 Identifiers

The environment ρ contains exactly the information that $\mathcal{P}$ returns for the identifier's abstract result and residual. Thus we only need to return a pair containing the current binding for the identifier and the function result environment.

$$\mathcal{P}[\![\texttt{ident}]\!]\rho\,\delta\,\xi\,d \;=\; \rho(\texttt{ident}) \tag{3.8}$$

3.5.3 Conditions

The evaluation of conditional expressions in the partial evaluation algorithm is somewhat more interesting than either constants or identifiers.

$$\mathcal{P}[\![(\texttt{if c } e_1\ e_2)]\!]\rho\,\delta\,\xi\,d = \tag{3.9}$$

let

$$< c^{\alpha}, c^{R} > = \mathcal{P}[\![c]\!]\rho\,\delta\,\xi\,d \tag{1}$$

$$< e^{\alpha}, e^{R} > = \begin{cases} \mathcal{P}[\![e_1]\!]\rho\,\delta\,\xi\,d & \text{if } \gamma(c^{\alpha}) = \texttt{true} \\ \mathcal{P}[\![e_2]\!]\rho\,\delta\,\xi\,d & \text{if } \gamma(c^{\alpha}) = \texttt{false} \\ < \bot, (\texttt{if c } e_1\ e_2) > & \text{if } \gamma(c^{\alpha}) = \bot \\ \mathcal{C}(c^{R}\ e_1\ e_2\ \rho\,\delta\,\xi) & \text{otherwise} \end{cases} \tag{2}$$

in

$$< e^{\alpha}, e^{R} > \tag{3}$$

end

The evaluation of a conditional begins with the evaluation of the controlling expression. We then have to decide whether to treat the resulting value as "static" or "dynamic". Recall that the intuitive meaning of "static" is "known at compile-time". In the case of a boolean expression, if the abstract value has the value "true" or "false" in the natural concrete domain, then we have definite knowledge about the value of the expression. Thus in line (2) we decide on our action based on $\gamma(c^\alpha)$, the value of the controlling expression when converted to the concrete domain. This is where our lifted and topped construction for concrete domains is used; if the result of concretizing the value yields $\top$ then we know that the abstract value cannot be assumed to represent exactly "true" or exactly "false". Further, this knowledge is independent of the abstraction chosen by the implementor of the abstract domain.

If we have exact knowledge then we can follow an evaluation that is very similar to that within the standard semantics — we simply evaluate the appropriate branch of the expression (the first two cases in line 2). Note that in this case the overall residual expression is simply the residual from the chosen branch; the actual `if` expression, the controlling expression, and the branch that is not chosen will not exist in the residual.

If we do not have complete knowledge of the result of the controlling expression then the overall result could be the result from either of the branches. Algorithm $\mathcal{C}$ deals with this evaluation and the construction of the appropriate residual.

$$
\begin{aligned}
&\mathcal{C}(\mathrm{c}^R\ \mathrm{e}_1\ \mathrm{e}_2\ \rho\,\delta\,\xi) = && (3.10)\\
&\quad \textbf{let}\\
&\qquad < \rho_T, \rho_F > = \mathit{Split}(\mathrm{c}^R, \rho) && (1)\\
&\qquad < \mathrm{e}_1^\alpha, \mathrm{e}_1^R > = \mathcal{P}[\![\mathrm{e}_1]\!]\rho_T\,\delta\,\xi\ \mathrm{true} && (2)\\
&\qquad < \mathrm{e}_2^\alpha, \mathrm{e}_2^R > = \mathcal{P}[\![\mathrm{e}_2]\!]\rho_F\,\delta\,\xi\ \mathrm{true} && (3)\\
&\quad \textbf{in}\\
&\qquad < \mathrm{e}_1^\alpha\ \nabla_P\ \mathrm{e}_2^\alpha,\ (\mathtt{if}\ \mathrm{c}^R\ \mathrm{e}_1^R\ \mathrm{e}_2^R) > && (4)\\
&\quad \textbf{end}
\end{aligned}
$$

If for the time being we ignore line 1, Algorithm C is fairly straightforward. The algorithm independently evaluates the two branches and creates the return value and residual. The abstract return value is the result of precisely widening the values from the evaluations of the branches. Intuitively one can think about this as a "union" operation expressing that the overall result is composed of any possible result from the branches. The residual is a new `if` statement composed of the residual of the controlling expression and the residual of each branch.

Algorithm C uses the additional helper, *Split*. Although it is always safe to interpret the branches of a conditional expression in the same environment as the entire statement, we would like to be able to take advantage of any implicit constraints present in the boolean expression that controls the branches. The *Split* routine takes a boolean conditional expression and a identifier binding environment and creates "true" and "false" resulting environments.

More formally, given a simple relational operation, $\theta_{\mathcal{D}}$ over a particular domain $\mathcal{D}$, and an environment in which identifier x is bound to a value in domain $\mathcal{D}$, then the following holds:

$$< \rho_T, \rho_F >= \mathit{Split}(\texttt{(x } \theta_{\mathcal{D}} \texttt{ y)}, \rho) \tag{3.11}$$

where the following three conditions hold:

Condition 1: `y` is a value in domain $\mathcal{D}$

Condition 2: for all $\varrho \sqsubseteq \rho$ such that $\mathcal{N}[\![\texttt{x } \theta_{\mathcal{D}} \texttt{ y}]\!]\varrho = \texttt{true}$ it is the case that $\varrho \sqsubseteq \rho_T$ and

Condition 3: for all $\varrho \sqsubseteq \rho$ such that $\mathcal{N}[\![\texttt{x } \theta_{\mathcal{D}} \texttt{ y}]\!]\varrho = \texttt{false}$ it is the case that $\varrho \sqsubseteq \rho_F$.

The "true" and "false" environments are created by modifying the original environment to take advantage of relationships expressed in the conditional expression. For example given a binding $\{\texttt{x} \mapsto [1 \ldots \infty]\}$ within environment ρ, a reasonable implementation of *Split*$(\texttt{(x < 5)}, \rho)$ would result in environments ρ_T and ρ_F where $\{\texttt{x} \mapsto [1 \ldots 4]\}$ would be the binding for x in ρ_T and $\{\texttt{x} \mapsto [5 \ldots \infty]\}$ would be the binding for x in ρ_F. Note that when y is an identifier rather than a value, the interpreter can perform transformations in order to evaluate the constraints for x and

y independently. The definition for *Split* is what the domains must provide; the interpreter uses this basic definition to perform more general forms of environment manipulation.

The general approach to implementing *Split* is to perform a simple abstract interpretation over conditional statements. In order to simplify the discussion at this point, we will use a trivial *Split* function that makes no improvements to the "true" and "false" scopes. We will discuss an actual implementation of *Split* in more detail in Section 6.2.

The definition for *Split* that we will assume is as follows:

$$Split(c, \rho) =< \rho, \rho > \tag{3.12}$$

In examples that we develop, we will generally assume that we have a slightly more accurate version of *Split*; any environment improvements that result will follow directly from simple conditional expressions. The proofs that are presented in Chapter 4 depend on only the properties of *Split*, the proofs do not depend on this particular definition of *Split*.

The interface to *Split* provides very little detail to the abstract domains. In particular, determining *all* of the potential constraints that might effect the resulting environments could potentially require that *Split* have access to the entire program and be able to interpret arbitrary program text. However, $\mathcal{P}$ already knows how to evaluate programs; $\mathcal{P}$ does not know how to manipulate abstract domain values. Thus the abstract domains perform simple abstract value splitting, while $\mathcal{P}$ performs the interpretation; an outline of this approach will be discussed in Section 6.2.

3.5.4 Function Properties

Primitive operators

Primitive operators are built-in to the source language. We require that for every primitive operator there exist a corresponding abstract version of the operator defined for the abstract domain. As noted earlier, we reuse the abstraction function α so that the abstract version of an operator is represented as $\alpha(\mathrm{op})$.

$$\mathcal{P}[\![(\mathrm{op}\ \ e_1\, e_2 \ldots e_n)]\!]\rho\,\delta\,\xi\,d = \tag{3.13}$$

let

$$< e_i^\alpha, e_i^R >= \mathcal{P}[\![e_i]\!]\rho\,\delta\,\xi\,d \quad \text{for all } 1 \le i \le n \tag{1}$$

$$v^\alpha = \mathit{apply}(\alpha(\mathrm{op}), e_1^\alpha\, e_2^\alpha \ldots e_n^\alpha) \tag{2}$$

$$v^R = \begin{cases} \gamma(v^\alpha) & \text{if } \gamma(v^\alpha) \notin \{\top, \bot\} \\ (\mathrm{op}\ \ e_1^R\, e_2^R \ldots e_n^R) & \text{otherwise} \end{cases} \tag{3}$$

in

$$< v^\alpha,\ v^R > \tag{4}$$

end

With respect to value computation, this evaluation is very similar to the corresponding operation in the standard semantics. The actual arguments are evaluated and the primitive operation is applied to the resulting abstract values. The interesting aspect of this part of the algorithm is in the construction of the residual. The basic decision is whether to leave the application of the primitive within the residual or to remove the application and to leave a simple result. The critical observation is that if we wish to eliminate the application, the value we place into the residual must be representable within the natural concrete domain. This makes the decision remarkably easy within our framework. We know that the result of applying γ to an abstract value yields either a particular value in the natural concrete domain or one of $\top$ or $\bot$, thus we can replace the application with a simple value exactly when $\gamma(v^\alpha)$ is not $\top$ or $\bot$.

For example, consider the expression `(+ 3 (if c 1 2))`. If we assume that `c`

is unknown, this corresponds to adding the value 3 to either 1 or 2. Even if the abstract domain is perfectly accurate and reflects the minimal set of values for `(if c 1 2)`, the best that the abstract operation $\alpha(+)$ could do for its result is to calculate a set of values including 4 and 5. Any concretization function for this set of values would return $\top$ since the natural integer domain in our language cannot express sets of values. Thus we would create a residual constructed from the "+" operator and the residuals of the two arguments. By applying the rule for constants (Equation 3.7) and the rule for dynamic conditional expressions (Equation 3.10), we determine that the overall residual is identical to the original expression. Note that using this particular model, we did not perform the algebraic manipulation of moving the addition operation into the `if` statement. Doing so would yield the residual `(if c 4 5)`, but such a transformation is beyond the scope of our current work. Note that in general such a transformation may not be desirable; in this case the only time such a transformation does make sense is if both the "3" and either the "1" or "2" evaluated to constants. If that did not hold then there would be duplication of the "3" expression which results in useless code expansion.

Function Application

In order to clarify the algorithm, we separate the general function application rule into two separate rules. The first rule deals with "unconditional" function applications which are the applications that will be evaluated in *any* evaluation of the given code under the standard semantics. As with the rule for primitive applications, the value manipulation mimics the behaviour of evaluation under the standard semantics; the argument is evaluated, a new environment binding the formal argument name to the abstract value is created, and the body of the function is evaluated in this context. The residuals for non-primitive function applications are created using exactly the same method as for primitive applications. If the abstract value can be safely transformed into a value in the corresponding natural concrete domain, then the value is representable in the residual and we replace the function application with the value. If the value cannot safely be represented, the application must re-

main in the residual.

$$\mathcal{P}[(\lambda x.e)\ e_1]\rho\,\delta\,\xi\ \text{false} = \tag{3.14}$$

$$\text{let}\ < e_1^{\alpha}, e_1^{R} > = \mathcal{P}[e_1]\rho\,\delta\,\xi\ \text{false} \tag{1}$$

$$x^{R} = \begin{cases} \gamma(e_1^{\alpha}) & \text{if } \gamma(e_1^{\alpha}) \notin \{\top, \bot\} \\ x & \text{otherwise} \end{cases} \tag{2}$$

$$< e^{\alpha}, e^{R} > = \mathcal{P}[e]\,\rho[x \mapsto < e_1^{\alpha}, x^{R} >]\,\delta\,\xi\ \text{false} \tag{3}$$

$$v^{R} = \begin{cases} \gamma(e^{\alpha}) & \text{if } \gamma(e^{\alpha}) \notin \{\top, \bot\} \\ (\lambda x.e^{R}\ e_1^{R}) & \text{otherwise} \end{cases} \tag{4}$$

in

$$< e^{\alpha}, v^{R} > \tag{5}$$

end

Using this rule, consider the following function application:

```
(define and
    (lambda (x y) (if x y false))
)
(and true z)
```

where z is unknown (has the value $\top$). Assume that the current identifier environment is empty. The evaluation of `(and true z)` begins by creating the identifier environment to be used for the evaluation of the body of the `and` function. By line 2, the residual for the formal argument `y` is bound to the identifier `y` because the value of the actual argument `z` is $\top$ and `x` is bound to the constant value `true`. We thus use the bindings {x↦<true,true>, y↦<$\top$,y>} for the evaluation of the function body. The identifier `x` has the value `true`, so we apply the rule for static conditionals (rule 3.9), resulting in the evaluation of the expression `y` only. Applying the identifier rule, the overall result for the body is <$\top$,y>. The actual parameter `z` has the value $\top$, so the second argument must remain in the residual, meaning that we use the second case for producing the residual. This choice results in the residual `((lambda (y) y) z)`. Assuming post-processing simplifications, this yields the overall result $< \top, z >$. This result means that although we do not know anything about the abstract value for the function application, we are able to simplify the residual for the expression to just the identifier `z`; the application of the function can be eliminated.

Dynamic Function Application

The second case for general function application covers the case of function evaluations that occur during the evaluation of a dynamic conditional expression.

$$\mathcal{P}[\![(\lambda x.\,e)e_1]\!]\rho\,\delta\,\xi\,\text{true} = \tag{3.15}$$

let

$$< e_1^a, e_1^R >= \mathcal{P}[\![e_1]\!]\rho\,\delta\,\xi\,\text{true} \tag{1}$$

in

if $e_1^a \sqsubseteq \delta(\lambda x.\,e_{id})$ then (2)

$$< \xi(\lambda x.\,e_{id}),\ (\lambda x.\,e\ e_1^R) > \tag{3}$$

else

let

$$\delta' = \delta[\lambda x.\,e_{id} \mapsto \delta(\lambda x.\,e_{id})\nabla_R\ e_1^a] \tag{4}$$

$$x^R = \begin{cases} \gamma(\delta'(\lambda x.\,e_{id})) & \text{if } \gamma(\delta'(\lambda x.\,e_{id})) \notin \{\top, \bot\} \\ x & \text{otherwise} \end{cases} \tag{5}$$

$$\rho' = \rho[x \mapsto < \delta'(\lambda x.\,e_{id}), x^R >] \tag{6}$$

$$< e^a, e^R >= \mathcal{P}[\![e]\!]\rho'\,\delta'\,\xi\,\text{true} \tag{7}$$

$$v^R = \begin{cases} \gamma(e^a) & \text{if } \gamma(e^a) \notin \{\top, \bot\} \\ (\lambda x.\,e^R\ e_1^R) & \text{otherwise} \end{cases} \tag{8}$$

$$\xi' = \xi[\lambda x.\,e_{id} \mapsto \xi(\lambda x.\,e_{id})\nabla_R\ e^a] \tag{9}$$

in

if $e^a \sqsubseteq \xi(\lambda x.\,e_{id})$ then (10)

$$< e^a,\ v^R > \tag{11}$$

else

$$\mathcal{P}[\![(\lambda x.\,e)e_1]\!]\rho\,\delta\,\xi'\,\text{true} \tag{12}$$

end

end

As discussed in Section 2.4.3, the on-line algorithm must make a decision about when to further investigate branches within a dynamic conditional expression. In our

algorithm, the decision about when to proceed further in the investigation is based on the search for fixed points in the series of function argument values and function return values.

This case is in many ways the heart of the entire on-line partial evaluation algorithm in that this case deals with potentially divergent function applications. Recall that at the beginning of Section 3.5, we introduced the δ and ξ environments. The δ environment maps function identifiers to estimates of argument values; the ξ environment maps function identifiers to estimates of result values.

This part of the algorithm begins in the expected way — simply evaluating the argument of the function. The guard in line 3.15(2) then checks whether the new argument is below the current estimate for this function in δ. If the current argument is below the argument estimate then we simply return our current result estimate.

If the algorithm has found a new parameter to this function, we must evaluate the function body with this new parameter. In order to guarantee that we make progress towards a safe estimate, we use the relaxed widening operator to extend the current estimate by the new parameter value. The widening operation may produce a new abstract value that represents arbitrarily more concrete values than the previous estimate. In order to produce a correct estimate of the function result, we must evaluate the body of the function with all of the new values. Thus, rather than simply using the abstract parameter value, we must use all of the new estimate (i.e. $\delta'(\lambda x.e_{id})$). In lines 3.15(5,6) the new identifier environment is built. Line 3.15(7) evaluates the body of the function using the new definitions. Lines 3.15(8,9) configure the function residual and an expanded result estimate.

The guard in line 3.15(10) determines whether the current value is new. If not, we can can produce this value and the residual as the result. If the result is new, we must continue our evaluation. It is important to note that in line 3.15(12) we must re-evaluate the entire original application, including the actual parameter. The reason for the full re-evaluation is that the actual parameter value may depend on the results of the function. If we do not re-evaluate the argument, the argument does not take the new result estimates into account.

3.5.5 An Example of the Algorithm

In order to illustrate the behaviour of the algorithm, we will consider a function that sums integers in the range from `start` to `stop`.

```
(define (Sum start stop)
   (if (> start stop)
       0
       (+ start (Sum (+ 1 start) stop))
) )
```

We will use the simple *negative/zero/positive* abstract domain that was discussed in Section 1.4.1. Within the traces we will simply use subsets of {N,Z,P} as our abstract values and use set union for both precise and relaxed widening. This example will be revisited in Section 5.1.3 but with a more accurate model for integers.

To reduce the effort needed to follow the examples, we have included a concise version of the algorithm (the rules only) in Appendix B.

In order to keep the example trace to a reasonable size, we will skip most of the "uninteresting" steps in the derivation and will focus on the recursive evaluations of `Sum`. In the example, we will evaluate `Sum` from `1` to `x` where `x` is unknown (i.e. $\top$). We assume that we have an accurate *Split* function.

Given an evaluation `(Sum (+ 1 start) stop)`, we will have a trace step of the form:

$$\texttt{(Sum } x\ y\texttt{)}\ \delta(start)\ \delta(stop)\ \xi$$

where x is the value of `(+ 1 start)`, y is the value of `stop`, and the δ and ξ values are as given. In terms of the evaluation, this captures the state of e_I^α for each argument and the state of δ and ξ immediately following line 3.15(1) where the actual parameter is evaluated.

Each nested evaluation of the body will be indented; since the re-evaluation of the entire expression with the new ξ environment (in line 3.15(12)) is strictly tail-recursive, we will not indent for this case. Since all but the initial call to `Sum` occur as a result of evaluating the body of `Sum`, after each completed recursive evaluation of `Sum` we will give the overall value for e^α in the form "$e^\alpha = Z \nabla_P (z + y)$". This reflects the basic evaluation for the body of `Sum` in any call for this example — the

conditional expression will always be unknown, so the overall result will always be a precise widening of the values of each branch. The value of the first conditional branch is always zero and $(z+y)$ is the value of the second conditional branch where z is the value of `start` during the evaluation of the body and y is the result of the recursive evaluation. It is very important to note that $z = \delta(start)\nabla_R\, x$ since, as defined by line 3.15(6), the body is evaluated in the ρ' environment found by widening the old δ value by the new e_1^α value.

Finally, after giving the new e^α value, we present a trace line that gives the value for ξ' which determines whether e^α is the result or whether another evaluation is necessary.

A sequence of trace lines from a recursive evaluation might look like the following:

```
(Sum P P) ⊥ ⊥ ⊥
      (Sum P P) P P ⊥
e^α = Z ∇_P (P + ⊥)
ξ' = ⊥ ∇_R Z
```

The two evaluated parameter values are given in the `(Sum P P)` fragment of the first line. In this example, it is not the case that both parameter values are below the respective values in δ (represented by the next two values in the trace line). Thus an evaluation of the body results. The evaluation of the body (eventually) yields another recursive evaluation of `Sum`.

The two evaluated parameter values for the recursive evaluation are given in the second `(Sum P P)` fragment. In this case each new parameter value is below the respective value in δ (the next two values in the trace line). This means that in the algorithm the value returned would be the value of ξ, which in this case is $\perp$.

The next trace line shows the computed value for the body of `Sum` for the first evaluation. Note that the "P" value in the expression "$(P + \perp)$" results from the value bound to `start` during the evaluation of the body. This value was calculated from a relaxed widening of the old δ value (i.e. $\perp$) by the e_1^α value (i.e. P).

The third line computes the new ξ' value which is always the old ξ value widened by the computed e^α value. In this case, the old ξ value is $\perp$ and the e^α value is Z. Since $e^\alpha \not\sqsubseteq \xi$, we must re-evaluate the original expression with the new ξ'.

(Sum P $\top$) $\bot$ $\bot$ $\bot$ (Note: d = false)

(Sum P P) $\bot$ $\bot$ $\bot$ (Note: d = true)

(Sum P P) P P $\bot$

$e^{\alpha} = Z \nabla_P (P + \bot)$

$\xi' = \bot \nabla_R Z = Z$

(Sum P P) $\bot$ $\bot$ Z

(Sum P P) P P Z

$e^{\alpha} = Z \nabla_P (P + Z)$

$\xi' = Z \nabla_R P = PZ$

(Sum P P) $\bot$ $\bot$ PZ

(Sum P P) P P PZ

$e^{\alpha} = Z \nabla_P (P + PZ)$

$\xi' = PZ \nabla_R PZ = PZ$

$Z \nabla_P (P + PZ)$

PZ

Due to the nature of the abstract domain, concretizing an abstract value other than "Z" does not produce a useful concrete value that can be used in a residual. Due to this loss of accuracy, when the form of a residual for a call is determined in lines 3.15(5,8), we will always produce trivial residuals. The overall residual is found to be a trivial single unfolding of the function. When more accurate abstract domains are introduced in Section 5.1.3, we will be able to make non-identity transformations to the program.

Chapter 4

Analysis of the On-line Algorithm

4.1 Derivations

In order to characterize computations under the standard semantics, we will need to be able to talk about derivations within computations in the standard semantics. A *derived evaluation* is simply an evaluation that is used as part of the evaluation of some other expression. We can give a somewhat informal inductive definition as follows:

1. `const` and `ident` have no derived expressions.
2. `(if c` e_1 e_2`)` has derived computations `c`, and either e_1 or e_2 depending on the value of `c`.
3. `(op` $e_1 e_2 \ldots e_n$`)` has derived computations $e_1 e_2 \ldots e_n$.
4. `(λx.e` e_1`)` has derived computations e_1 and `e[x` $\mapsto r$`]` where r is the value of e_1.

It follows directly from the definition of derivations that the evaluation of $\mathcal{N}[\![E]\!]\varrho$ must have a finite number of derived evaluations if the evaluation is well-defined. Thus there are two conditions under which $\mathcal{N}[\![E]\!]\varrho$ is not defined — if the result of a non-total primitive is not defined or if $\mathcal{N}[\![E]\!]\varrho$ does not have a finite derivation.

4.2 Soundness and Termination

The algorithm that we have presented for partial evaluation performs an interpretation of the original program. The interpretation algorithm is not guaranteed to terminate in *all* cases. In particular, if every possible execution of the source program diverges, the interpreter will diverge. Recall that in Chapter 2 we introduced static and dynamic partitions of data in the context of the *Mix* equations. The first *Mix* equation expresses the idea that we incorporate the static data into the original program to produce a new program that executes with the dynamic data. A (slightly modified) version of the first *Mix* equation is as follows:

$$\mathcal{N}[\![e]\!](s, D) = \mathcal{P}[\![(e, s)]\!](D)$$

In terms of this definition, we will show that given an expression e and static data s where there exists dynamic data D such that $\mathcal{N}[\![e]\!](s, D)$ is well-defined, then $\mathcal{P}[\![(e, s)]\!]$ is well-defined. The partial evaluation algorithm *may* terminate even if there is no D such that $\mathcal{N}[\![e]\!](s, D)$ is well-defined, but we do not formally guarantee termination in such cases. In our approach we treat the static data s as being embedded in e; this is manifested in that neither $\mathcal{P}$ nor $\mathcal{N}$ take a static environment as a parameter.

Recall that the dynamic data D is encapsulated in the initial environments for $\mathcal{N}$ and $\mathcal{P}$. In the case of $\mathcal{P}$ all such bindings will be to $\top$ while for $\mathcal{N}$ all bindings will be to particular concrete data. The identifiers mapped by the initial environments for either $\mathcal{N}$ or $\mathcal{P}$ are always the same — the free variables in the expression being evaluated. We can apply the concept of ordering to these environments; in particular we can define a $\sqsubseteq$ operator for the environments. We define $\sqsubseteq$ between identifier environments ϱ and ρ as follows: $\varrho \sqsubseteq \rho$ if for all identifiers $x \in \varrho$, there exists a mapping $(x \mapsto v) \in \rho$ such that $\alpha(\varrho(x)) \sqsubseteq \mathit{first}(v)$. This means that an environment is "below" a second environment if all of the bindings contained in the first environment are below the bindings in the second environment. Note that we will be using quantifiers over the ϱ environments used in evaluating an expression e with $\mathcal{N}$. Such quantifications relate to the values bound to the free variables in e. For example, the statement "for all $\varrho \ldots$" should be interpreted as "for any set of concrete values bound to the identifiers in $\varrho \ldots$".

The definition for "below" for two function argument environments, δ and δ' is more straightforward: $\delta \sqsubseteq \delta'$ if for all bindings $\lambda x.e_{id} \mapsto x^{\alpha} \in \delta$ it is the case that $\delta(\lambda x.e_{id}) \sqsubseteq \delta'(\lambda x.e_{id})$.

Finally, we define "below" for two function return environments in the same way as for function argument environments: $\xi \sqsubseteq \xi'$ if for all bindings $\lambda x.e_{id} \mapsto \lambda x.e^{\alpha}_{id} \in \xi$ it is the case that $\xi(\lambda x.e_{id}) \sqsubseteq \xi'(\lambda x.e_{id})$.

As noted in Section 3.3.1, we use the term "below" to mean "below or equal to" and use the term "strictly below" for the stronger relationship.

Proof of Soundness

We begin by proving several useful properties that will assist in the main proof. The basic idea of the first theorem is that if an abstract value can be concretized into a single concrete value, the concretized value must be the same as the original value.

Theorem 4.1

Given any abstract value e^{α} where $\gamma(e^{\alpha}) \notin \{\top, \bot\}$ and $\mathcal{N}[\![e]\!]\varrho \sqsubseteq \gamma(e^{\alpha})$ then $\mathcal{N}[\![e]\!]\varrho = \gamma(e^{\alpha})$.

Proof:

By assumption $\gamma(e^{\alpha}) \notin \{\top, \bot\}$ so there exists a value $c = \gamma(e^{\alpha})$ in the concrete domain. Assume that there is some value c' such that $c \neq c'$ and $\mathcal{N}[\![e]\!]\varrho = c'$. Then $c' \sqsubseteq c$. The definition of concrete domain states that given x, y in some concrete domain C such that $x, y \notin \{\top, \bot\}$ and $x \not\parallel y$ then $x = y$. This contradicts $c \neq c'$, so it must be the case that $\mathcal{N}[\![e]\!]\varrho = \gamma(e^{\alpha})$.

□

The next theorem only refers to $\mathcal{N}$ and characterizes the nature of recursive evaluations in the standard semantics. The basic point of the theorem is that if we have some set of correct solutions for a function f, then for any evaluation of f that produces a new result, there is some recursive evaluation of f that produces a new result without relying on another recursive evaluation of f to produce a result outside the estimate. A more intuitive way of stating this property is that any evaluation that produces a new result does so with only a finite number of recursive calls that themselves produce new results.

The important thing to note with respect to this theorem is that the theorem is not simply a finiteness property; the fundamental statement is that given a new

argument value there must be some particular evaluation that only relies on either known results of the function, or on no further recursive evaluation of the function. This property will be critical in proving our main theorem.

Theorem 4.2

Given sets of values D, D', and X, such that $D \subseteq D'$ and $r \in D$ implies $\mathcal{N}[\![\lambda x.\, e\ r]\!]\varrho \in X$ then if there exists $r_1 \in D'$ such that

1. *$\mathcal{N}[\![\lambda x.\, e\ r_1]\!]\varrho$ is defined and*
2. *$\mathcal{N}[\![\lambda x.\, e\ r_1]\!]\varrho \notin X$*

then there exists $r^* \in D'$ such that $\mathcal{N}[\![\lambda x.\, e\ r^*]\!]\varrho \notin X$ does not derive any $\mathcal{N}[\![\lambda x.\, e\ r_2]\!]\varrho$ where $r_2 \in D'$ and $\mathcal{N}[\![\lambda x.\, e\ r_2]\!]\varrho \notin X$.

Proof:

Assume $r_1 \in D'$ such that $\mathcal{N}[\![\lambda x.\, e\ r_1]\!]\varrho \notin X$. Either $\mathcal{N}[\![\lambda x.\, e\ r_1]\!]\varrho$ derives some $\mathcal{N}[\![\lambda x.\, e\ r_2]\!]\varrho'$ such that $r_2 \in D'$ and $\mathcal{N}[\![\lambda x.\, e\ r_2]\!]\varrho \notin X$ or it does not derive any such evaluation.

Assume that for all $r_1 \in D'$, $\mathcal{N}[\![\lambda x.\, e\ r_1]\!]\varrho$ derives some $\mathcal{N}[\![\lambda x.\, e\ r_2]\!]\varrho'$ such that $r_2 \in D'$ and $\mathcal{N}[\![\lambda x.\, e\ r_2]\!]\varrho \notin X$. Then each $r_1 \in D'$ derives some other value in D' and the derivation of $\mathcal{N}[\![\lambda x.\, e\ r_1]\!]\varrho$ can not be not finite.

Thus there exists $r^* \in D'$ such that $\mathcal{N}[\![\lambda x.\, e\ r^*]\!]\varrho \notin X$ and $\mathcal{N}[\![\lambda x.\, e\ r^*]\!]\varrho$ does not derive any $\mathcal{N}[\![\lambda x.\, e\ r_2]\!]\varrho$ where $r_2 \in D'$ and $\mathcal{N}[\![\lambda x.\, e\ r_2]\!]\varrho \notin X$.

□

All of the following proofs of soundness and termination only rely on properties of the abstract values that $\mathcal{P}$ produces. In order to simplify the proofs slightly, we will ignore the residuals that are produced. In terms of notation, this means that we will allow direct comparisons such as $\mathcal{P}[\![e]\!]\,\rho\,\delta\,\xi\,d \sqsubseteq A$ for an abstract value A rather than the full expression $\mathit{first}(\mathcal{P}[\![e]\!]\,\rho\,\delta\,\xi\,d) \sqsubseteq A$.

We will also be somewhat lazy with respect to one additional aspect of our notation. The statements of the theorems relate values found by $\mathcal{N}$ to values found by $\mathcal{P}$. In order to have a meaningful $\sqsubseteq$ relationship, we must convert the values produced

by $\mathcal{N}$ into the abstract domain using the abstraction function α. Rather than repeating the α on every comparison, we adopt the additional convention that a comparison such as $\mathcal{N}[\![e]\!]\varrho \sqsubseteq A$ for some abstract value A will mean $\alpha(\mathcal{N}[\![e]\!]\varrho) \sqsubseteq A$.

The following theorem is our main theorem; the actual soundness statement is a direct corollary of this theorem. The basic statement of the theorem is that given some set of correct result estimates for some set of argument values, the result of $\mathcal{P}$ is correct when evaluating any expression.

The theorem is quantified over the environment given to $\mathcal{P}$; the theorem holds for an environment that is a safe estimate for some set of possible environments used in the standard semantics.

There are two basic preconditions for the theorem:

1. the environments that we consider are those for which the expression is well-defined in the standard semantics, (i.e. those for which the derivation is finite and does not produce bottom), and
2. that for any function, f, the result estimates in $\xi(f)$ are correct for the arguments in $\delta(f)$.

Theorem 4.3

For all expressions E, $\varrho \sqsubseteq \rho$, *and boolean values* d *such that*

1. $\mathcal{N}[\![E]\!]\varrho$ *is defined, and*
2. *for all functions* f *and expressions* e_1 *derived by* $\mathcal{N}[\![E]\!]\varrho$,

$$\alpha(\mathcal{N}[\![e_1]\!]\varrho) \sqsubseteq \delta(\lambda x.e_{id}) \textit{ implies } \alpha(\mathcal{N}[\![f\ e_1]\!]\varrho) \sqsubseteq \xi(\lambda x.e_{id})$$

it is the case that

$$\alpha(\mathcal{N}[\![E]\!]\varrho) \sqsubseteq \mathit{first}(\mathcal{P}[\![E]\!]\rho\,\delta\,\xi\ d).$$

Using both of the shortcuts in notation, the theorem can be re-stated as the following:

For all expressions E, $\varrho \sqsubseteq \rho$, and boolean values d such that

1. **$\mathcal{N}[\![\mathtt{E}]\!]\,\varrho$ is defined, and**
2. **for all functions f and expressions $\mathtt{e}_1$ derived by $\mathcal{N}[\![\mathtt{E}]\!]\,\varrho$,**

$$\mathcal{N}[\![\mathtt{e}_1]\!]\varrho \sqsubseteq \delta(\lambda \mathtt{x}.\mathtt{e}_{id}) \text{ implies } \mathcal{N}[\![\mathtt{f}\ \ \mathtt{e}_1]\!]\varrho \sqsubseteq \xi(\lambda \mathtt{x}.\mathtt{e}_{id})$$

it is the case that

$$\mathcal{N}[\![\mathtt{E}]\!]\varrho \sqsubseteq \mathcal{P}[\![\mathtt{E}]\!]\,\rho\,\delta\,\xi\,d.$$

Proof:

Case 1: $\mathtt{E} = \mathtt{const}$.

E is evaluated by $\mathcal{N}$ using rule 3.2 and $\mathcal{P}$ applies rule 3.7. By definition of rule 3.2, for any ϱ, $\mathcal{N}[\![\mathtt{const}]\!]\,\varrho = \mathtt{const}$. By definition of rule 3.7, $\mathcal{P}[\![\mathtt{const}]\!]\,\rho\,\delta\,\xi\,d = \alpha(\mathtt{const})$. Thus, by definition of α,

$$\mathcal{N}[\![\mathtt{const}]\!]\,\varrho \sqsubseteq \mathcal{P}[\![\mathtt{const}]\!]\,\rho\,\delta\,\xi\,d.$$

Case 2: $\mathtt{E} = \mathtt{ident}$.

E is evaluated by $\mathcal{N}$ using rule 3.3. Then $\mathcal{P}$ applies rule 3.8. Since by assumption $\varrho \sqsubseteq \rho$ we have $\varrho(\mathtt{ident}) \sqsubseteq \rho(\mathtt{ident})$.

Case 3: $\mathtt{E} = \mathtt{(if\ c\ e1\ e2)}$.

E is evaluated by $\mathcal{N}$ using rule 3.4. Then $\mathcal{P}$ applies rule 3.9. In both 3.4(1) and 3.9(1) the subexpression c is evaluated.

By induction, $\alpha(\mathtt{c}) \sqsubseteq \mathtt{c}^{\alpha}$.

The cases in 3.9(2) depend on whether $\gamma(\mathtt{c}^{\alpha}) \in \{\top, \bot\}$.

Case i: Assume $\gamma(\mathtt{c}^{\alpha}) \in \{\mathtt{true},\ \mathtt{false}\}$. By Thm. 4.1, the value of c' in 3.4(1) must be the same as $\gamma(\mathtt{c}^{\alpha})$. Since $\gamma(\mathtt{c}^{\alpha}) = c'$, we know that $\mathcal{N}$ and $\mathcal{P}$ evaluate the same subexpression of E in 3.4(2) and 3.9(2) respectively. Thus, by induction, the result of $\mathcal{N}$ is below the result produced by $\mathcal{P}$.

Case ii: Assume $\gamma(\mathtt{c}^{\alpha}) = \top$. Then Algorithm $\mathcal{C}$ (3.10) is applied, and $\mathcal{P}$ evaluates both $\mathtt{e}_1$ and $\mathtt{e}_2$ producing $\mathtt{e}_1^{\alpha}$ and $\mathtt{e}_2^{\alpha}$.

Assume $\mathcal{N}$ evaluates only e_1, producing e_1'. Consider the evaluation of e_1 in rule 3.10. By definition of *Split*, the ρ_T environment produced by *Split* must be above any ϱ such that condition c is satisfied. Thus if e_1 is evaluated by $\mathcal{N}$, $\varrho \sqsubseteq \rho_T$. This satisfies the conditions for induction, so $\alpha(e_1') \sqsubseteq e_1^\alpha$.

Since by induction e_1^α satisfies the theorem and by definition $x \sqsubseteq x \nabla_P y$ for all x, y, we have $\alpha(e_1') \sqsubseteq e_1^\alpha \sqsubseteq (e_1^\alpha \nabla_P e_2^\alpha)$ and thus by transitivity $\alpha(e_1') \sqsubseteq (e_1^\alpha \nabla_P e_2^\alpha)$, so the theorem holds.

A symmetric argument holds when $\mathcal{N}$ evaluates only e_2.

Case iii: Assume $\gamma(c^\alpha) = \bot$. Then $\mathcal{N}[\![c]\!]\varrho = \bot$. This means that no evaluation of $\mathcal{N}[\![c]\!]\varrho$ is defined, which contradicts our theorem assumption. Having $\mathcal{P}$ produce $\bot$ is consistent with the result from $\mathcal{N}$ — the evaluation does not have a defined meaning.

Case 4: $E = (\mathrm{op}\ e_1 e_2 \ldots e_n)$.

E is evaluated by $\mathcal{N}$ using rule 3.5. Then $\mathcal{P}$ applies rule 3.13.

By definition of $\alpha(\mathrm{op})$, we know that if $\alpha(e_i) \sqsubseteq e_i^\alpha$ for all $1 \leq i \leq n$ then

$$\mathit{apply}(\mathrm{op}\ e_1 e_2 \ldots e_n) \sqsubseteq \mathit{apply}(\alpha(\mathrm{op})\ e_1^\alpha e_2^\alpha \ldots e_n^\alpha)$$

In rules 3.5 and 3.13 respectively, each of the subexpressions e_i is evaluated. By induction, the result of $\mathcal{N}$ is below the result produced by $\mathcal{P}$ for each subexpression e_i. Thus by definition the result of applying op in $\mathcal{N}$ must be below the result of applying $\alpha(\mathrm{op})$ in $\mathcal{P}$. Thus the theorem holds.

Case 5: $E = (\lambda x.e\ e_1)$ and $d = \mathrm{false}$.

E is evaluated by $\mathcal{N}$ using rule 3.6. Then $\mathcal{P}$ applies either rule 3.14 or rule 3.15. Assume $\mathcal{P}$ applies rule 3.14.

The overall result from $\mathcal{P}$ is e^α. In order to show that the theorem holds, we only need to show that $\varrho[x \mapsto e_1'] \sqsubseteq \rho[x \mapsto < e_1^\alpha, e_1^R >]$ in order to apply induction. By our inductive hypothesis, $\varrho \sqsubseteq \rho$ so for all identifiers other than x, $\alpha(\varrho(\mathrm{ident})) \sqsubseteq \rho(\mathrm{ident})$. By induction $\alpha(e_1') \sqsubseteq e_1^\alpha$, so $\alpha(\varrho(x)) = \alpha(e_1') \sqsubseteq e_1^\alpha$. Thus the theorem holds.

Case 6: $E = (\lambda x.e\ e_1)$ and $d = \mathrm{true}$.

The final case is when a function call is evaluated by $\mathcal{N}$ using rule 3.6 and by $\mathcal{P}$ using rule 3.15.

There are three main cases to the proof; each case corresponds to one of the return values generated by $\mathcal{P}$ (lines 3.15(3,11,12)). Case 1 and Case 3 are the straightforward cases, while Case 2 has a more interesting behaviour.

Case 1 holds when the value of the given argument is already in our set of possible argument values. Since $\xi(\lambda x.\, e_{id})$ is the set of solutions for the given arguments, it is safe to return the current result estimate (in $\xi(\lambda x.\, e_{id})$).

Case 2 is the most interesting case. In this case we have not discovered any new results even though we have new possible arguments. The key to this case is in showing that there cannot be any result in the standard semantics that is in fact outside the current estimate.

Case 3 is the case when we have discovered new function results. In this case, we must re-evaluate the function taking into account the new results.

The following holds by simple induction and is used in each case, so we state it before going into the three cases.

$$\mathcal{N}[\![e_1]\!]\,\varrho \sqsubseteq \mathcal{P}[\![e_1]\!]\,\rho\,\delta\,\xi\,\text{true} \tag{4.16}$$

Case i: $\mathcal{P}[\![e_1]\!]\,\rho\,\delta\,\xi\,\text{true} \sqsubseteq \delta(\lambda x.\, e_{id})$.

This case holds when the value of the given argument is already in our set of possible argument values. Since $\xi(\lambda x.\, e_{id})$ is the set of solutions for the given arguments, it is safe to return the current result estimate (in $\xi(\lambda x.\, e_{id})$).

By transitivity of $\sqsubseteq$, given the case assumption and 4.16, $\mathcal{N}[\![e_1]\!]\,\varrho \sqsubseteq \delta(\lambda x.\, e_{id})$. Thus by the second precondition of the theorem, $\mathcal{N}[\![\lambda x.\, e\ e_1]\!]\,\varrho \sqsubseteq \xi(\lambda x.\, e_{id})$. By assumption of the case, the guard in line 3.15(2) of the algorithm holds, so by line 3.15(3),

$$\mathcal{P}[\![\lambda x.\, e\ e_1]\!]\,\rho\,\delta\,\xi\,\text{true} = \xi(\lambda x.\, e_{id}). \tag{4.17}$$

Thus by substitution (from 4.17),

$$\mathcal{N}[\![\lambda x.\, e\ e_1]\!]\,\varrho \sqsubseteq \mathcal{P}[\![\lambda x.\, e\ e_1]\!]\,\rho\,\delta\,\xi\,\text{true}.$$

Case ii: $\mathcal{P}[\![e_1]\!]\,\rho\,\delta\,\xi\,\text{true} \not\sqsubseteq \delta(\lambda x.e_{id})$ and $\mathcal{P}[\![e]\!]\,\rho'\,\delta'\,\xi\,\text{true} \sqsubseteq \xi(\lambda x.e_{id})$.

In this case we have not discovered any new results even though we have new possible arguments. We must show that it cannot be the case that $\mathcal{P}$ produces a result below $\xi(\lambda x.e_{id})$ if we would have new results from any evaluation in the standard semantics. This is a proof by contradiction; we assume that $\mathcal{N}$ produces a result outside of $\xi(\lambda x.e_{id})$ and show that a contradiction results.

Assume $\mathcal{N}[\![(\lambda x.e\ e_1)]\!]\,\varrho \not\sqsubseteq \xi(\lambda x.e_{id})$. By the contravariant of the second theorem precondition, it follows that $\mathcal{N}[\![e_1]\!]\,\varrho \not\sqsubseteq \delta(\lambda x.e_{id})$.

By construction (line 3.15(4)), $\delta \sqsubseteq \delta'$. Since the second precondition of the theorem holds, by Theorem 4.2 there must exist some $r^* \sqsubseteq \delta'(\lambda x.e_{id})$ such that

$$\mathcal{N}[\![\lambda x.e\ r^*]\!]\,\varrho \not\sqsubseteq \xi(\lambda x.e_{id}) \tag{4.18}$$

where

$$\mathcal{N}[\![\lambda x.e\ r^*]\!]\,\varrho \text{ does not derive any } \mathcal{N}[\![\lambda x.e\ r_2]\!]\,\varrho \not\sqsubseteq \xi(\lambda x.e_{id}) \tag{4.19}$$

Now consider the evaluation $e^\alpha = \mathcal{P}[\![e]\!]\rho'\,\delta'\,\xi\,\text{true}$ in line 3.15(7).

By construction, $\rho' = \rho[x \mapsto \delta']$. By assumption, $r^* \sqsubseteq \delta'(\lambda x.e_{id})$, so $\varrho[x \mapsto r^*] \sqsubseteq \rho'$. By 4.19 (from Theorem 4.2), all computations derived from $\mathcal{N}[\![\lambda x.e\ r^*]\!]\,\varrho$ satisfy the second precondition of the theorem. Thus by induction, it is the case that

$$\mathcal{N}[\![\lambda x.e]\!]\,\varrho[x \mapsto r^*] \sqsubseteq \mathcal{P}[\![e]\!]\rho'\,\delta'\,\xi\,\text{true}.$$

By assumption, $\mathcal{N}[\![\lambda x.e\ r^*]\!]\,\varrho \not\sqsubseteq \xi(\lambda x.e_{id})$ so $\mathcal{P}[\![e]\!]\rho'\,\delta'\,\xi\,\text{true} \not\sqsubseteq \xi(\lambda x.e_{id})$. But this contradicts our case assumption.

Thus, $\mathcal{N}[\![(\lambda x.e\ e_1)]\!]\,\varrho \sqsubseteq \xi(\lambda x.e_{id})$.

Case iii: $\mathcal{P}[\![e_1]\!]\,\rho\,\delta\,\xi\,\text{true} \not\sqsubseteq \delta(\lambda x.e_{id})$ and $\mathcal{P}[\![e]\!]\,\rho'\,\delta'\,\xi\,\text{true} \not\sqsubseteq \xi(\lambda x.e_{id})$.

In this the case we have discovered new function results. We must re-evaluate the function taking into account the new results.

By definition of the algorithm, in this case neither of the guards in lines 3.15(2) and 3.15(10) hold, so the result is

$$\mathcal{P}[\lambda x.\, e\ e_1]\,\rho\,\delta\,\xi'\ \text{true}.$$

By construction (line 3.15(4)), $\xi(\lambda x.\, e_{id}) \sqsubseteq \xi'(\lambda x.\, e_{id})$ so by transitivity of $\sqsubseteq$,

$$\mathcal{N}[\lambda x.\, e\ e_1]\,\varrho \sqsubseteq \xi(\lambda x.\, e_{id}) \text{ implies } \mathcal{N}[\lambda x.\, e\ e_1]\,\varrho \sqsubseteq \xi'(\lambda x.\, e_{id}).$$

Thus by induction,

$$\mathcal{N}[\lambda x.\, e\ e_1]\,\varrho \sqsubseteq \mathcal{P}[\lambda x.\, e\ e_1]\,\rho\,\delta\,\xi'\ \text{true}.$$

□

Corollary 4.3.1 *($\mathcal{P}$ is Sound)*

For all expressions E, $\varrho \sqsubseteq \rho$, *and boolean values* d *such that* $\mathcal{N}[\![E]\!]\,\varrho$ *is defined it is the case that* $\mathcal{N}[\![E]\!]\varrho \sqsubseteq \mathcal{P}[\![E]\!]\,\rho\,\delta\,\xi\,d$ *where* $\delta(f_{id}) = \perp$ *for all* f_{id} *and* $\xi(f_{id}) = \perp$ *for all* f_{id}.

Proof:

This follows trivially from Theorem 4.3 since $\perp$ is a correct result for any expression in $\mathcal{N}$ given $\perp$ as an argument.

□

Termination

We will set the proofs of termination in the context of the derivations introduced earlier, except that we now label each step in the derivation. In addition, we now introduce derivations for $\mathcal{P}$. In order to distinguish between derivations in $\mathcal{N}$ and $\mathcal{P}$, we will call use the term *evaluation path* when talking about derivations in $\mathcal{P}$.

A (possibly infinite) *evaluation path* is a sequence $\mathcal{P}_1, \ldots$ representing the steps in a derivation by $\mathcal{P}$. Each $\mathcal{P}_i$ represents an application of some rule in the algorithm of the form $\mathcal{P}[\![e_i]\!]\rho_i\,\delta_i\,d_i$. We will use the additional notations $\rho_1, \ldots$, and $\delta_1, \ldots$, and

$d_1, \ldots$ to represent the respective sequences of parameters to evaluations $\mathcal{P}_1, \ldots$ in some evaluation path.

The overall proof of termination is constructed from three lemmas. The first lemma shows that if the d parameter to $\mathcal{P}$ becomes *true*, it remains *true*. The second lemma shows that the the number of steps that $\mathcal{P}$ takes with d being *false* is bounded by the number of steps taken in an evaluation by $\mathcal{N}$. The third lemma shows that the number of steps taken by $\mathcal{P}$ when d is *true* is bounded.

Lemma 4.3.1 *Given an evaluation path $\mathcal{P}_1, \ldots$ and k such that $d_k = true$ then there exists j such that $0 \leq j < k$ and $d_1, \ldots, d_j, \ldots$ is of the form $false_1, \ldots, false_j, true_{j+1} \ldots$.*

Proof:

Given an evaluation $\mathcal{P}_i$, then either $d_i = d_{i-1}$ (by rules 3.9(1,2), 3.13(1), 3.14(1,3)), and 3.15(1,7,12)) or $d_i = \text{true}$ (by rules 3.10(2,3)). Thus, by induction, for any $i \leq j$ we have $d_j = \text{false}$ implies $d_i = \text{false}$ and for any $i \geq j + 1$ we have $d_{j+1} = \text{true}$ implies $d_i = \text{true}$.

□

Lemma 4.3.1 does not deal with the value of the d parameter for the base case. The two cases are straightforward: it follows from the Lemma that if $d_1 = \text{false}$ then $j \geq 1$ and if $d_1 = \text{true}$ then $j = 0$.

Lemma 4.3.2 *Given an evaluation path $\mathcal{P}_1, \ldots$ for $\mathcal{P}[\![\mathrm{E}]\!]\rho\,\delta\,d$ and a finite evaluation path $\mathcal{N}_1, \ldots, \mathcal{N}_n$ for $\mathcal{N}[\![\mathrm{E}]\!]\varrho$ where $\varrho \sqsubseteq \rho$ then the number of $\mathcal{P}_i$ evaluations with $d_i = false$ is less than or equal to the n.*

Proof:

We prove this lemma by showing that for an evaluation of E by $\mathcal{N}$, if $d = \text{false}$ then $\mathcal{P}$ performs no more evaluation steps than $\mathcal{N}$.

Case 1: E = const.

Then each of $\mathcal{N}$ and $\mathcal{P}$ return a value in a single step.

Case 2: E = ident.

Then each of $\mathcal{N}$ and $\mathcal{P}$ return a value in a single step.

Case 3: E = (if c e_1 e_2).

Then E is evaluated by $\mathcal{N}$ using 3.4 and by $\mathcal{P}$ using 3.9.

In both 3.4(1) and 3.9(1) the subexpression c is evaluated. By induction we assume that $\mathcal{P}$ takes no more steps than $\mathcal{N}$.

The cases in 3.9(2) depend on whether $\gamma(c^\alpha) \in \{\top, \bot\}$.

Case i: Assume $\gamma(c^\alpha) \in \{$true, false$\}$.

By Thm. 4.1, the value of c' in 3.4(1) must be the same as $\gamma(c^\alpha)$. Since $\gamma(c^\alpha) = c'$, $\mathcal{N}$ and $\mathcal{P}$ evaluate the same subexpression of E in 3.4(2) and 3.9(4) respectively. By induction we conclude that $\mathcal{P}$ takes no more evaluation steps than $\mathcal{N}$.

Case ii: Assume $\gamma(c^\alpha) = \bot$.

Then $\mathcal{P}$ returns $\bot$ in a single step and the theorem holds.

Case iii: Assume $\gamma(c^\alpha) = \top$.

Then Algorithm $\mathcal{C}$ (3.10) is applied, and the evaluations of e_1 and e_2 have $d =$ true. Thus $\mathcal{P}$ evaluates neither e_1 nor e_2 with $d =$ false while $\mathcal{N}$ evaluates one of e_1 and e_2. Thus $\mathcal{P}$ takes fewer steps with $d =$ false than the number of steps taken by $\mathcal{N}$.

Case 4: E = (op e_1 $e_2 \ldots e_n$).

Then E is evaluated by $\mathcal{N}$ using rule 3.5 and by $\mathcal{P}$ using rule 3.13. In each rule each subexpressions e_i is evaluated. By induction we assume that $\mathcal{P}$ takes no more steps than $\mathcal{N}$ on each argument, so the theorem holds.

Case 5: E = (λx.e e_1).

Then E is evaluated by $\mathcal{N}$ using rule 3.6 and by $\mathcal{P}$ using rule 3.14 since by assumption $d =$ false.

By induction, the evaluations by $\mathcal{P}$ in lines 3.14(1,3) must take fewer steps than the corresponding evaluations by $\mathcal{N}$ in lines 3.6(1,2). Thus the theorem holds.

□

Lemma 4.3.3 *Given an expression E and an evaluation path with d_1 = true then there exists n such that the sequence that $\mathcal{P}_1, \ldots$ terminates at $\mathcal{P}_n$.*

Proof:

By Lemma 4.3.1, d_i = true for all $i > 1$ since by assumption d_1 = true. By definition of rules 3.14 and 3.15 this implies that all function applications in $\mathcal{P}_1, \ldots$ are evaluated using rule 3.15.

Let $f_1, f_2, \ldots, f_m$ be the finite universe of function identifiers evaluated by $\mathcal{P}_1, \ldots$. Consider the sequence $\delta_1, \delta_2, \ldots$ of function argument environments.

By definition of the algorithm, for all steps other than 3.15(4), we have $\delta_{i+1} = \delta_i$. In 3.15(4), we have that $\delta_{i+1} = \delta_i[\lambda x.e_{id} \mapsto (\delta_i(\lambda x.e_{id}) \ \nabla_R \ e_1^a)]$. Since $\delta_{i+1} = \delta_i$ holds for all rules other than 3.15(2), we now ignore the other steps in the evaluation and consider only the sequence $\mathcal{P}_1, \ldots$ where each $\mathcal{P}_i$ is an evaluation of a function application using rule 3.15.

We first show that after a finite number of steps the sequence $\delta_1, \ldots$ reaches a fixed-point; i.e. that $\delta_{k+1} = \delta_k$ for some k. We will then show that if $\delta_{k+1} = \delta_k$ then $\mathcal{P}_{k+1}$ terminates.

Consider the separate sequences of values mapped to f_i within $\delta_1, \ldots$. For each f_i we label the sequence as $x_1^{f_i}, x_2^{f_i}, \ldots$. For simplicity, we'll consider a single sequence $x_1, x_2, \ldots$ for a given function f_i. There are two possibilities for the sequence $x_1, \ldots$: if f_i is evaluated only m times, then for all $j > 0$, $x_{m+j} = x_m$. If f_i is evaluated an unbounded number of times, then by 3.15(4), $x_{i+1} = x_i \ \nabla_R \ v_i$ where v_i is the value of e_1^a found by $\mathcal{P}_i$ in line 3.15(1). By definition of ∇_R, for any function f and value x_0, there exists a k such that $f(x_k) \sqsubseteq x_k$ where $x_i = x_{i-1} \nabla_R f(x_{i-1})$ for $i > 0$.

Since by definition of 3.15(4) and 3.15(1), $v_i = g(v_{i-1})$ where "g" is evaluated in 3.15(1), there must exist a k such that $x_{k+1} = x_k \ \nabla_R \ v_k = x_i$. Since our argument was made for any function f_i, we know that for each function f_i in $f_1, \ldots, f_m$ there exists

a corresponding k_i. By assumption, $f_1, \ldots, f_m$ is finite, so after at most $K = \sum k_i$ steps, we know that $\delta_{K+1} = \delta_K$.

We have left to show that if $\delta_{k+1} = \delta_k$ then $\mathcal{P}_{k+1}$ terminates. By definition of rule 3.15(6) the abstract values bound to identifiers in sequence of ρ environments follow the same widening operations as the δ sequence. Thus when $\delta_{k+1} = \delta_k$ we also will have $\rho_{k+1} = \rho_k$ if we consider only the abstract values in each ρ_i (i.e. we ignore the residuals in each ρ_i). If $\rho_{k+1} = \rho_k$ and $\delta_{k+1} = \delta_k$ then $v_{k+1} = \mathcal{P}[\![e_1]\!]\rho_{k+1}\,\delta_{k+1}$ true $= \mathcal{P}[\![e_1]\!]\rho_k\,\delta_k$ true $= v_k$. But the fixed-point of the δ environments is found with respect to the sequence of v_i values, so $v_{k+1} \sqsubseteq \delta_{k+1}(f_i)$. Thus the guard in line 3.15(10) is satisfied and 3.15(11) produces a value.

Theorem 4.4 *($\mathcal{P}$ terminates)*

> *Given an expression* E *such that* $\mathcal{N}[\![\mathrm{E}]\!]\,\varrho$ *terminates for some environment* ϱ*, then given any* $\rho \sqsupseteq \varrho$, $\mathcal{P}[\![\mathrm{E}]\!]\,\rho\,\delta\,\xi\,d$ *terminates.*

Proof:

By Lemma 4.3.2, any sequence of evaluations $\mathcal{P}_1, \ldots$ in which every $d_i = \mathit{false}$ must be finite. By Lemma 4.3.1 if there exists some $d_i = \mathit{true}$ then for all $j > i$, we know that $d_j = \mathit{true}$. Finally, by Lemma 4.3.3, any sequence of evaluations in which all $d_j = \mathit{true}$ must be finite. Thus the entire evaluation must be finite.

□

Lemmas 4.3.1 and 4.3.3 are interesting in terms of the "behaviour" of the overall algorithm. By selecting d_1 = false we realize the termination statement we have given, but a corollary of Lemma 4.3.3 is that selecting d_1 = true results in an algorithm that guarantees termination in *all* cases. Although in practice such a choice results in a substantial loss of accuracy, this observation leads to an heuristic for guaranteeing termination in all cases — allow d_1 = false, but select a value j such that for any $i > j$ the interpreter forces d_j = true. This method for termination forces the abstract domains to find fixed-points over all calls in an evaluation path *including* the static function calls.

4.3 Correctness of Residuals

The correctness results in the previous section ignored the residuals; although we now know that the abstract values are sound with respect to calculations in the standard semantics and that the abstract calculation terminates, we still need to show that the residuals calculate the same result as any interpretation in the standard semantics. The argument is a straightforward structural induction over any expression and relies on the soundness results from the previous section.

Theorem 4.5 *($\mathcal{P}$ produces correct residuals)*

> *Given an expression* E *such that* $e^R = \text{second}(\mathcal{P}[\![E]\!]\,\rho\,\delta\,\xi\,d)$ *then for all* $\varrho \sqsubseteq \rho$, *such that* $second(\rho(\texttt{ident}))$ *is a correct residual for* ident, *it is the case that* $\mathcal{N}[\![e^R]\!]\varrho = \mathcal{N}[\![\texttt{E}]\!]\varrho$.

Proof:

Case 1: E = const.

E is evaluated using by $\mathcal{N}$ using 3.2 and by $\mathcal{P}$ using 3.7. Since const is the residual, it must be correct.

Case 2: E = ident.

E is evaluated using by $\mathcal{N}$ using 3.3 and by $\mathcal{P}$ using 3.8. By assumption, $second(\rho(\texttt{ident}))$ is a correct residual for ident.

Case 3: E = (if c e_1 e_2).

Then E is evaluated by $\mathcal{N}$ using 3.4 and by $\mathcal{P}$ using 3.9.

In both 3.4(1) and 3.9(1) the subexpression c is evaluated. By induction we assume that c^R is correct. The overall residual produced in this case depends on the which choice is made in 3.9(2).

Case i: Assume $\gamma(c^a) =$ true.

By Thm. 4.3.1, every evaluation in the standard semantics produces true for c. Thus the original expression is equivalent to (if true e_1 e_2). By definition of the standard semantics (line 3.4(2)),

the result of this expression is the result of evaluating e_1 in the same environment as the original expression. Thus it is safe to evaluate only e_1. Since by induction e_1^R is a safe residual for e_1, e_1^R is a safe residual for the entire expression. A symmetric argument holds when $\gamma(c^\alpha) = \texttt{false}$.

Case ii: Assume $\gamma(c^\alpha) = \bot$.

Then the original expression is the residual. Trivially this is a safe residual.

Case iii: Assume $\gamma(c^\alpha) = \top$.

Then Algorithm $\mathcal{C}$ (3.10) is applied, and $\mathcal{P}$ evaluates both e_1 and e_2 producing residuals e_1^R and e_2^R. Since by induction each of the residuals for the subexpressions are correct and since $\mathcal{C}$ simply replaces each component of the overall expression with correct subexpressions, the residual produced by $\mathcal{C}$ must be correct.

Case 4: $\texttt{E} = \texttt{(op } e_1\ e_2 \ldots e_n\texttt{)}$.

Then E is evaluated by $\mathcal{N}$ using rule 3.5 and by $\mathcal{P}$ using rule 3.13. In each rule each subexpressions e_i is evaluated. By induction we assume that the residual for each subexpression is correct.

There are two cases for the construction of the residual.

Case i: $\gamma(v^\alpha) \notin \{\top, \bot\}$.

Then by Thm. 4.1, $\gamma(v^\alpha)$ is exactly the value produced by $\mathcal{N}[\![\texttt{E}]\!]\rho$, so we can trivially replace the operation by this value.

Case ii: $\gamma(v^\alpha) \in \{\top, \bot\}$.

Then the residual is the original operation applied to the residuals of the arguments. Since the residual of each argument is correct, the entire residual is correct.

Case 5: $\texttt{E} = \texttt{(}\lambda\texttt{x.e } e_1\texttt{)}$ and $d = \text{false}$. Then E is evaluated by $\mathcal{N}$ using rule 3.6 and by $\mathcal{P}$ using rule 3.14 since by assumption $d = \text{false}$.

We need to show that new residual bindings created in ρ are correct and that the overall residual is correct.

Part 1: The residual bound to x within ρ is either $\gamma(e_1^\alpha)$ or the identifier x itself. If $\gamma(e_1^\alpha)$ is chosen as the residual then $\gamma(e_1^\alpha) \notin \{\top, \bot\}$ and by Thm. 4.1,

this is a correct residual. If the identifier x is chosen as the residual then the binding is correct assuming that x is a formal parameter in the final residual. By definition of 3.14(4), x will be a formal parameter in the final residual unless the final residual is a constant in the concrete domain. If the final residual is a constant, then no identifiers can exist in the residual in which case any residual for x in ρ would be trivially correct.

Part 2: There are two cases for the overall residual. If $\gamma(e_1^a)$ is chosen as the residual then $\gamma(e_1^a) \notin \{\top, \bot\}$ and by Thm. 4.1, this is a correct residual. If $(\lambda\lambda x.e^R\ e_1^R)$ is the overall residual then the overall residual must be correct since by induction, e^R and e_1^R are both correct.

Case 6: $E = (\lambda x.e\ e_1)$ and $d = \text{true}$.

Then E is evaluated by $\mathcal{N}$ using rule 3.6 and by $\mathcal{P}$ using rule 3.15 since by assumption $d = \text{true}$.

By induction e_1^R is correct, so the residual produced in 3.15(3) is correct.

By the a similar argument as in Case 5 (Part 1), the residual bound to x in ρ is correct. If the residual in line 3.15(11) is the result, then by the same argument as in Case 5 (Part 2), the residual must be correct. If the residual in line 3.15(12) is the result, then since the bindings passed to the recursive evaluation are the same as those passed to this call, by induction the resulting residual must be correct.

□

There a few interesting points to note about this proof. First, in Case 3(i), we take advantage of the fact that our language is pure. This is used by appealing to the definition in the standard semantics in which the environment for the subexpressions is the same as the environment for the original expression. If the conditional were permitted to cause side-effects within the environment we would have to modify our approach. In order to make any non-trivial transformation in such cases, we would have to determine if the conditional *actually* contains impure computations. Such a computation could be made by using a two part abstraction domain in which we consider "may-alias" [53] [29] information as part of the abstract domain. This would require fairly small changes to the interpreter. We could then use this alias information to create a residual for the conditional that causes the same side-effect as the

evaluation of the original expression. By performing this transformation we would in fact remove the original `if` expression and replace it with a sequential evaluation of the residual for the conditional (i.e. the code causing the side-effect) followed by the code for the appropriate branch. Any may-alias analysis would be a conservative approximation since the existence of aliases is undecidable in general. However, a may-alias analysis is a relatively simple form of abstract interpretation and thus would fit nicely into our approach.

4.4 On the Efficiency of On-line Evaluation

There are two main factors that determine the complexity of on-line partial evaluation. The first factor is the cost of operations in the abstract domains; the second factor is the overhead imposed by the evaluation algorithm itself. In our approach, the partial evaluation algorithm is parameterized by the abstract domains and any restriction on the running time of the abstract operations would restrict potential domains that an implementor may want to use. A complete evaluation of the complexity of evaluation given an arbitrary program has not been made. Although there are some aspects of the analysis that are reasonably straightforward, there are non-trivial interactions between the algorithm being evaluated, the abstract domain definitions, and the accuracy of the environments used when evaluating branches in a conditional statement. We will use the term *well-behaved* to mean that if the evaluation of e by $\mathcal{N}$ requires at most $O(g(n))$ primitive operations for any input, then $\mathcal{P}$ requires at most $O(g(n))$ abstract domain operations.

The first observation is that the on-line algorithm is well-behaved when no conditionals are dynamic. The proof that $\mathcal{P}$ is well-behaved when no conditionals are dynamic has essentially already been given — Lemma 4.3.2 shows that whenever all conditionals are static, every evaluation step in $\mathcal{N}$ has a corresponding evaluation step in $\mathcal{P}$.

Given that the static analysis is well-behaved, we now characterize some of the potential difficulties that can be encountered after a dynamic conditional statement.

Consider the following functions:

```
(define complex
    (lambda (x)
        (if (< x 10) 1
            (if (< x 5) (ackermans x) 1)
)   )   )
(define f
    (lambda (x)
        (if (< x 10) 1
            (if (< x 20) (f (- x 1))
                         (f (- x 5)))
)   )   )
```

Function `complex` executes in $O(1)$ time for all input, while function `f` is $O(n)$. Unfortunately, whether our algorithm discovers these facts is dependent on the *Split* operation over integer domains. If, during the evaluation of `(< x 5)` in function `complex`, the *Split* operation retains the information that `x` must be greater than or equal to `10`, then $\mathcal{P}$ operates in $O(1)$ time for all input as well. However, given our trivial identity implementation of *Split* as presented earlier, this information would be lost and the algorithm would investigate the `ackermans` function — a very expensive choice. With function `f` the situation is even worse; there is a dependency between the evaluation of the second recursive call and the first. If the evaluator does not recognize the dependency, the partial evaluation of `f` will require exponential time since the algorithm investigates each branch of an `if` statement on each recursive call. This choice makes the partial evaluation of function `f` an exponential time evaluation.

In general, the fact that abstract interpreters investigate multiple branches of a conditional when the standard semantics requires these branches to be mutually exclusive is the cause of the exponential time behaviour. Termination is not the issue; the amount of work to achieve termination is. Most abstract interpretations, such as the early *negative/zero/positive* example, have abstract values that are in some finite (and generally very shallow) lattice. This means that even though exponential behaviour can be experienced, the exponent is bounded by a very small constant (the height of the lattice). Using the domain requirements that we have given, there are guaranteed bounds in our approach as well, but the bounds are dependent on the speed of convergence for the ∇_R operators. However, the domain convergence rate

is not the only factor — issues such as accuracy and memoization can change the effective complexity of the basic algorithm.

The problem encountered with function `complex` is not that difficult to handle; we could simply require that domains provide accurate *Split* information. Accurate *Split* information would guarantee that we would never investigate a conditional branch unless it could possibly be evaluated for some real input.

If we assume that the implementation of $\mathcal{P}$ performs memoization, then the particular problem with function `f` can be handled as well. After the forward analysis through the first recursion, a memoization of `f` can be created. This memoized version of `f` would, by the soundness theorem, have abstract parameter values that cover at least the full range of potential values for `x` along that branch. Thus when we eventually investigate the second recursive branch, this memoized version of `f` will be available for re-use within the second nested evaluation and the first recursive call will not be re-evaluated.

Finally, an additional phase could be introduced into the algorithm. This phase would analyze each function definition and determine whether there is more than one path through the function to a recursive call of the function. If more than one such path exists then the potential exponential behaviour could be avoided by using more traditional shallow, fixed-height lattices for that portion of the analysis. This type of technique is commonly used to ensure that harmful code duplication does not occur. Examples of harmful code duplication include causing redundant computation and duplicating code that contains operations with side-effects. Although these particular issues are discussed further in Sections 6.3 and 6.3.3, the application of these approaches to controlling exponential behaviour has not been investigated in any approach.

In the implementation developed as part of this work, false exponential behaviour has not been observed. We believe that this is due to the combination of having accurate domains and memoization. It remains as future work to determine an exact characterization of the system interactions that would formally guarantee a well-behaved partial evaluation algorithm.

4.5 Parameterizing Partial Evaluation

There has only been one other substantial investigation into parameterizing partial evaluation. In [25], Consel and Khoo report on a *facet* based approach to parameterizing partial evaluation. Their basic approach is to define algebras that relate the abstract domains to the concrete domains. They then investigate a simple on-line partial evaluator and off-line binding time analysis using their algebras. The major restriction in their approach is that they assume finite-height lattices for the abstract domains. Although they make the observation that a Cousot and Cousot type of widening operator would admit infinite-height lattices into the model, this approach has not been investigated further. In [20], Colby and Lee directly implement Consel and Khoo's approach. They observe that structured domains cannot be abstracted in a very expressive manner due to the restrictions of the abstract domains.

Our approach differs in that we explicitly admit infinite-height lattices with specifications as to required operations on such domains. In addition, we characterize both precise and imprecise abstract value operations and use the precise operations whenever termination can be insured. This differs from traditional approaches that solely use least-upper bounds for collecting abstract information.

Consel and Khoo basically ignore termination issues by leaving the decision about unfolding to the interpreter at the time that a specialization is performed. Their outline for an on-line partial evaluator abstracts away this entire decision by using an application function, APP, that determines whether to continue specialization or not. In Colby and Lee's implementation, the APP function makes this choice based solely on the depth of the inlining.

The most directly comparable work in terms of the proof framework is the work by Khoo and Consel [24] that forms the basis for their parameterized system. Their approach is to define a set of logical relations that relate an instrumented semantics, an on-line evaluator, and an off-line evaluator. The main proofs deal with correctness of the correspondence between the various semantics. They do not formally prove any form of termination condition, but as in the parameterized system, rely solely on decisions by the specializer to determine termination. Although the formal approach characterizes the specialization decisions as a *filter* function that monotonically increases to a fixed-point, the formal approach does not address how to deal with non-finite height domains. This is partially evident in the fact that structured

domains are not addressed. The bias of Khoo and Consel's work is to investigate the relationship between on-line and off-line evaluation and to formally characterize off-line binding time analysis. Their work effectively relates off-line binding time analysis (both monovariant and polyvariant) to forms of on-line evaluation, but is not as expressive for on-line partial evaluation as the approach proposed in this thesis.

As noted in Section 2.3.5, the FUSE system is a larger implementation effort than either this work or the work by Khoo and Consel, but the analytic side of the FUSE work does not address the relationship between abstract and standard semantics and depends on a finite height lattice model for termination properties.

In our approach, the model for termination is related to the abstract domains; the basic intuition is that unfolding can only be profitable if we are learning new information. We do not claim that this is a sufficient condition for useful unfolding, but at least at the partial evaluation level, it is necessary. In other words, without taking into consideration size of code, delay slots, and other "back-end" issues, we can only determine the usefulness of code inlining based on information that we are encountering. Although our basic criteria can be implemented in Consel and Khoo's model (as can any model), in order to have a reasonable compromise between accuracy and termination, it is important to differentiate between abstract value collections that effect termination and those that do not. Such a differentiation cannot be made in Consel and Khoo's model since their only method for collecting abstract values is by using least-upper bounds in a finite height lattice describing the domains. Since our approach separates the types of collections into precise and relaxed widenings, where only the relaxed widenings effect termination, we can more accurately manipulate the abstract information.

4.6 Summary of the On-line Framework

As we have seen in this chapter, the on-line algorithm that we have developed is dependent on only a few characteristics of the actual abstract domains chosen to represent information during the evaluation. The algorithm itself uses precise analysis whenever it can guarantee that divergence will not occur; while the accuracy of results is dependent on the accuracy of the abstract domains, the correctness is dependent on only the few required characteristics. The three phases in the on-line algorithm allow the *interpreter* to make the choice about when to switch the type of

analysis and to use as much of the information about the state of the analysis as possible. Combining the analysis and specialization phases presents opportunities for further optimizations and fits well with most standard approaches for producing good residuals; this will be discussed further throughout Chapter 6. The proofs that we have presented rely only on the basic characteristics of the abstract domains chosen to model information. This approach allows one to consider the design of flexible models for information as a problem that is nearly independent of the actual evaluation algorithm.

Chapter 5

Domain Implementations

5.1 Integer Interval Domains

When presenting the formal partial evaluation algorithm, we assumed that we had domains and widening operations for various basic types. In this section we will carefully introduce an abstract domain and corresponding widening operators for representing integers. There are several parts to this process: the definition of the abstract domain, the definition of the widening operators, and finally, a proof that the operators satisfy the requirements for precise and relaxed widening given in Definitions 3.1 and 3.2. As was observed during the discussion of the roles of domains in Section 3.3, these definitions and proofs are sufficient to demonstrate the correctness of the abstract evaluator with respect to integer values. In Section 5.2 we will follow the same process for the structured domain of Scheme lists.

5.1.1 Definition of Integer Interval Domains

> **Defn 5.1 (Integer Interval)** *An* integer interval, *V, is a sequence of integers $[x_1..x_n]$ such that $x_1 \leq x_n$. An interval* contains *all integers in the range; $\forall k, x_1 \leq k \leq x_n \Longrightarrow k \in V$.*

An interval with only a single integer in the range may be represented without the brackets. Given two intervals, V_1 and V_2, we will say that $V_1 \preccurlyeq V_2$ if $V_1 = \bot$ or $V_2 = \top$

or if $\forall k,\ k \in V_1 \Longrightarrow k \in V_2$. Integer intervals form a partial order with respect to $\preccurlyeq$. The intuitive meaning for $\top$ and $\bot$ is that $\top$ is the unbounded interval (contains all of N and $\bot$ is the empty interval (containing no elements). Two intervals may be ordered by a $<$ operation if all of the elements in one interval are less than all elements in the other interval; i.e. $V_1 < V_2$ if $\forall x \in V_1, \forall y \in V_2\ x < y$.

Theorem 5.1 *Integer Interval Lattice*
The integer intervals *are a complete lattice under* $\preccurlyeq$.

Proof:

We need to show that for any subset S of integer intervals, both $\bigvee S$ and $\bigwedge S$ exist.

Part 1: $\bigvee S$
Let s^{max} and s^{min} respectively be the minimum and maximum integer in the intervals in S. Then $v = [s^{min}..s^{max}]$ is an upper bound for S since for any interval $s \in S$, $s \preccurlyeq v$. We now need to show that v is the *least* upper bound. Assume there is some other upper bound $v' \prec v$. Then by definition of integer interval and $\prec$, $v' = [v'_1..v'_2]$ where either $v'_1 > s^{min}$ or $v'_2 < s^{max}$ (or both). Assume $v'_1 > s^{min}$. Then there exists a set $s \in S$ such that $s^{min} \in s$ and $s^{min} \notin v'$. Thus $s \not\preccurlyeq v'$ and v must be the least upper bound. A symmetric argument holds when $v'_2 < s^{max}$.

Part 2: $\bigwedge S$
Let S' be a set of integer sets where each set represents the elements of a corresponding interval in S. Let $v = \bigcap S'$.

Claim: v is representable as an interval. If v is empty or represents a single integer, v can be expressed as an interval. If v consists of several elements, then *each* interval in S must contain all of those elements. If v does not represent a contiguous series of integers then there exists some x, y, z with $x, z \in v$ and $y \notin v$ such that $x < y < z$. By construction this implies that there is some $s \in S'$ such that $x, z \in s$ and $y \notin s$. But this contradicts the definition of an interval, so v must be representable as an interval.

Claim: $v = \bigwedge S$. First we show that v is a lower bound. By construction, $\forall s \in S, \forall x \in v, x \in s$. Thus by definition, $v \preccurlyeq s$. Next we must show that v is the greatest lower bound. If v is not the greatest lower bound, then there exists some $v' \succ v$ such that $\forall s \in S, \forall x \in v', x \in s$. If $v' \succ v$ then there exists an element $x \in v'$ such that $x \notin v$ and

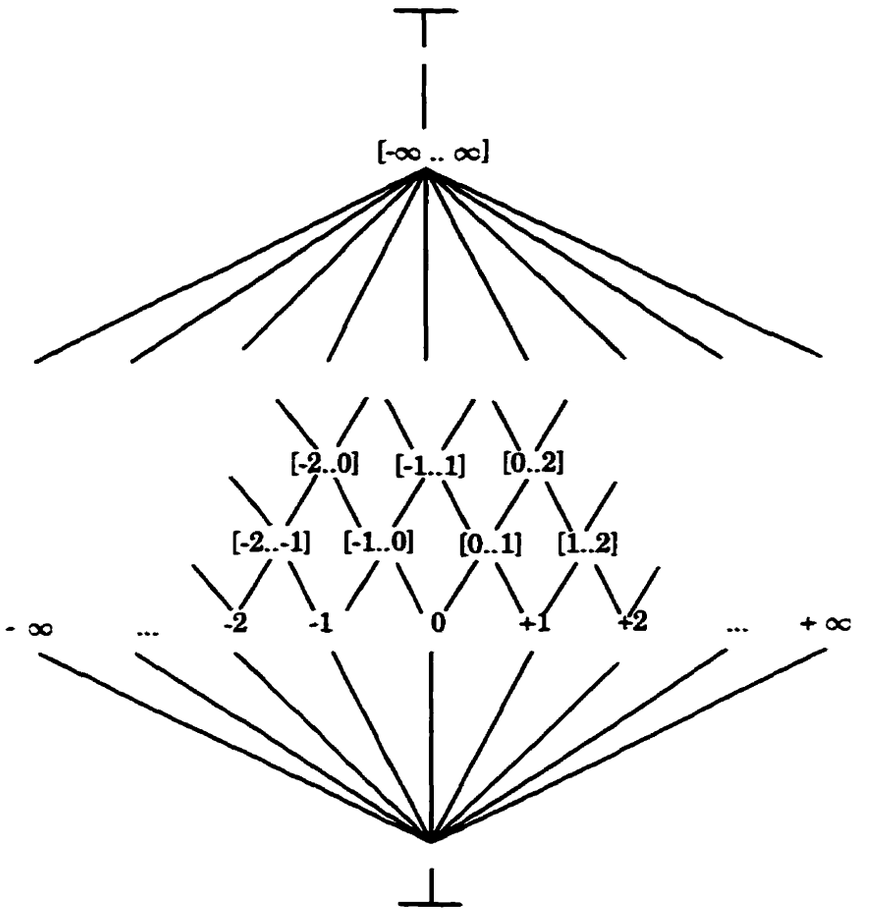


Figure 5.1.1: Integer Interval Lattice

$\forall s \in S, x \in s$. But if $\forall s \in S, x \in s$ then x is in the intersection of the sets in S' and then by definition, $x \in v$. Thus v must be the greatest lower bound.
□

Next, we need to define widening operations on integer intervals that preserve the nature of the domains. We will use two additional relationships between intervals to assist in these definitions. Two intervals *conjoin* if their values overlap or are immediately next to each other. The formal definition will provide us with a method of indicating that two intervals can be merged to form a larger single interval that contains strictly the elements in the two original intervals. Note that the idea of conjoining intervals is only needed in the definition of the widening operators; no integer interval domain will be allowed to contain a pair of conjoining intervals.

Defn 5.2 (Conjoint Intervals) *Let* $V_1 = [a..b]$ *and* $V_2 = [c..d]$ *be two integer intervals.* $V_1 \rightsquigarrow V_2$ *if* $c > a$ *and* $c \leq b + 1$ *and* $d > b$. *If* $V_1 \rightsquigarrow V_2$, *we say that* V_1 conjoins V_2.

Observation 5.1 *Given intervals* V_1 *and* V_2, *if* $V_1 \preccurlyeq V_2$ *then* $\neg(V_1 \rightsquigarrow V_2)$.

This observation follows directly from the definitions of $\preccurlyeq$ and conjoint intervals. If $V_1 \preccurlyeq V_2$ then all elements in V_1 are in V_2, but the definition of conjoint requires that the smallest value in V_1 is not in V_2.

Defn 5.3 (Disjoint Intervals) *Let* V_1 *and* V_2 *be two integer intervals. We say that* V_1 *and* V_2 *are* disjoint, *or symbolically that* $V_1 \leftrightharpoons V_2$ *if:*

1. $\neg(V_1 \rightsquigarrow V_2)$ *and* $\neg(V_2 \rightsquigarrow V_1)$ *and*
2. *for all* k, $k \in V_1 \Longrightarrow k \notin V_2$ *and* $k \in V_2 \Longrightarrow k \notin V_1$.

This definition of disjoint is a bit stronger than normal definitions; not only can the intervals not share any values, but there must be a "gap" between the elements. More formally, there must exist some x such that $V_1 < [x..x] < V_2$ or $V_2 < [x..x] < V_1$. For any pair of intervals, (V_1 and V_2), either the intervals are related by inclusion ($V_1 \preccurlyeq V_2$ or $V_2 \preccurlyeq V_1$), are conjoint ($V_1 \rightsquigarrow V_2$ or $V_2 \rightsquigarrow V_1$), or are disjoint ($V_1 \leftrightharpoons V_2$).

The conjoining formalism will be used to indicate when we will be able to merge a series of intervals into a single interval. For example, the intervals [1..10] and [11..20] are conjoint, as are [1..10] and [5..20]. In each of these cases we could replace the pair of intervals with a single interval [1..20] which would represent exactly the same values as the pair of intervals. [1..10] and [12..20] are disjoint since there is a "gap" between the two intervals; replacing these intervals with the single interval [1..20] would introduce an additional element, 11, that is not present in either of the original intervals.

We now extend the idea of two conjoining intervals to a series of conjoining intervals which we will call a *conjoining chain*. A conjoining chain is simply a series of intervals in which each interval conjoins the next one in the chain.

Defn 5.4 (Conjoining Chain) *Let* $V_1, V_2, \ldots, V_n$ *be integer intervals.* $CC(V_1, V_2, \ldots, V_n)$ *holds if* $\forall i : \{1..n-1\}, V_i \rightsquigarrow V_{i+1}$. *If* $CC(V_1, V_2, \ldots, V_n)$, *we say* $V_1, V_2, \ldots, V_n$ *are a* conjoining chain.

Defn 5.5 (Integer Interval Domain) *A value in an* integer interval domain, D^I *is a series of intervals* $\{V_1 < V_2 < \ldots < V_n\}$ *such that* $\forall V_i, V_j \in D^I : V_i \rightleftharpoons V_j$. *For a particular integer value,* x, *we say that* $x \in D^I$ *if there exists* $V \in D^I$ *such that* $x \in V$. $\perp$ *is considered to be in any value of an integer interval domain.*

The given domain descriptions define a particular normalization of sets of integers: *intervals* are ordered, contiguous subsets of integers and *domain values* are formed from an ordered set of disjoint intervals. This normalization is important as it allows for reasonable implementation; we could ignore implementation issues and simply define the integer abstraction as sets of integers, but in real implementations, manipulating arbitrary sets becomes very expensive.

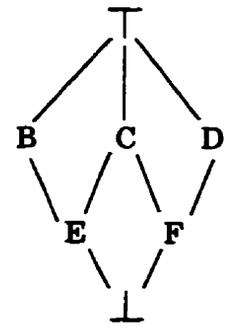


Figure 5.1.2: Abstract Value Covering

Recall that in Section 3.3.1, we defined the $\sqsubseteq$ operator for abstract domains in terms of the atoms in the down-sets of abstract domain values. For the integer interval domain, we can easily define a $\sqsubseteq$ operator that meets this requirement. Given values V_1 and V_2 in the integer interval domain, we say that $V_1 \sqsubseteq V_2$ if for every $x \in V_1$ there exists $y \in V_2$ such that $x \preccurlyeq y$. This statement may seem trivial, but does not

necessarily hold in abstract domains that are not normalized as the integer intervals are. To generalize this statement to any abstract domain, the statement implies that there can be no pair of abstract elements in V_2 that "cover" a single abstract element in V_1. Figure 5.1.2 gives an example of this type of cover — using the atoms in the down-set, we require that $\{C\} \sqsubseteq \{B, D\}$ since the atoms are the same, but clearly C is not below either B or D.

5.1.2 Widening Operators for Integer Intervals

We now define the widening operators, ∇_P^I and ∇_R^I, for the integer interval domains. In order to clarify this section, we will assume that the ∇_P and ∇_R operators will implicitly reference the integer interval domain. The definitions are a bit laborious as they must preserve the desired normalization in the representation.

Defn 5.6 (Precise Integer Widening) *Given integer domain values* $D_1 = \{V_1, V_2, \ldots, V_n\}$ *and* $D_2 = \{W_1, W_2, \ldots, W_m\}$,

$$(D_1 \nabla_P \ D_2) = \begin{cases} \{\top\} & \text{if } D_1 = \{\top\} \text{ or } D_2 = \{\top\} \quad \textit{(i)} \\ D_1 & \text{if } D_2 = \{\bot\} \quad \textit{(ii)} \\ D_2 & \text{if } D_1 = \{\bot\} \quad \textit{(iii)} \\ D_1 \nabla_P \ (D_2 - \{W_j\}) & \textit{(iv)} \\ \quad \textit{where } \exists V_i \in D_1 : W_j \sqsubseteq V_i & \\ (D_1 - \{V_i\}) \nabla_P \ D_2 & \textit{(v)} \\ \quad \textit{where } \exists W_j \in D_2 : V_i \sqsubseteq W_j & \\ D_1 \bigcup D_2 & \textit{(vi)} \\ \quad \textit{where } \forall V_i \in D_1, W_j \in D_2 : V_i \leftrightharpoons W_j & \\ (D_1 - \{V_i, \ldots, V_{i+k}\} + \{[a..b]\}) \nabla_P & \textit{(vii)} \\ \qquad (D_2 - \{W_j, \ldots, W_{j+p}\}) & \\ \quad \textit{where } CC(\{V_i, \ldots, V_{i+k}\} \bigcup \{W_j, \ldots, W_{j+p}\}), & \\ \qquad a = \textit{min}(\{V_i, W_j\}), \textit{ and} & \\ \qquad b = \textit{max}(\{V_{i+k}, W_{j+p}\}) & \end{cases}$$

The meaning of the definition is more intuitive than the definition itself might lead one to believe. The interval $\top$ acts as the multiplicative zero for widening: $\forall x, (\{\top\} \nabla_P\ x) = (x \nabla_P\ \{\top\}) = \{\top\}$ (Case i). The interval $\bot$ acts as the identity value for widening: $\forall x, (\{\bot\} \nabla_P\ x) = (x \nabla_P\ \{\bot\}) = x$ (Cases ii, iii). If there is an interval in one domain that is below an interval in the other domain, the lower interval is removed (Cases iv, v). If all intervals in the two domains are disjoint then the result is in its minimal form (Case vi). Finally, if there is a set of intervals that form a conjoining chain, those intervals are merged into a single interval in the result (Case vii).

Theorem 5.2 *(∇_P produces an Interval Domain value)*

Given integer domains D_1 and D_2, the result of $D_1 \nabla_P\ D_2$ is an integer interval domain value.

Proof:

The definition of ∇_P is inductive; at each step the definition either produces an interval domain value or reduces the number of intervals in the domain values by at least one. Thus we use a simple induction to show that the result is correct.

Cases (i) through (iii), and (vi) form the basis for the induction since they do not recursively apply ∇_P . Cases (i) through (iii) are trivially correct. Case (vi) is correct since all intervals are disjoint. In Cases (iv) and (v) we reduce the size of one of the domains, so by induction the result is correct. In Case (vii) we reduce the total number of intervals by at least one, so by induction this produces an interval domain value.

□

Theorem 5.3 *(∇_P is Precise)*

Precise integer widening (Defn. 5.6) satisfies the conditions for precise widening (Defn. 3.1).

Proof:

There are two conditions that must be satisfied for ∇_P to satisfy Defn. 3.1. First, if $V = V_1 \nabla_P\ V_2$ then $\downarrow V = (\downarrow V_1) \bigcup (\downarrow V_2)$. Second $\forall x, y \in V : x \neq y \Longrightarrow x \| y$.

We will first deal with the incomparability requirement. By definition of an interval domain values and Thm. 5.2, we know that there no intervals x and y in V such that $x \preccurlyeq y$. Thus, by definition of incomparability, $\forall x, y \in V : x \neq y \Longrightarrow x \| y$).

We now prove that if $V = V_1 \nabla_P V_2$ then $\downarrow V = (\downarrow V_1) \bigcup (\downarrow V_2)$. Recall that $\downarrow V$ is the set of atoms below the elements of V. We show that every transformation step in Defn. 5.6 preserves the set of atoms in the original domains. Case (i), (ii), and (iii) are trivial. In Cases (iv) and (v), the element being removed is below an element that is being preserved so the set of atoms is unchanged. In Case (vi) no elements are being removed so the set of atoms is unchanged. Finally, in Case (vii) we compress a conjoining chain into a single interval. In any conjoining chain the set of atoms is simply the set of integer values represented by the chain. Since the new interval reflects exactly these elements, the overall set of atoms is the same.

□

Precise interval widening works as one might expect – it creates the smallest set of intervals that contain exactly the information present in either of the domains. For example:

$$\{[-10..5], [7..11]\} \nabla_P \{[-11..-1], [1..4], [13..13]\} = \{[-11..5], [7..11], [13..13]\}.$$

The transformations are a straightforward application of the definition. [1..4] is below [−10..5] so by part (iv) of the rule, we remove [1..4] leaving $\{[-10..5], [7..11]\} \nabla_P \{[-11..-1], [13..13]\}$. Intervals [−11..−1] and [−10..5] are conjoint, so we merge them by part (vii) into a single interval, leaving $\{[-11..5], [7..11]\} \nabla_P \{[13..13]\}$. Since these intervals are all disjoint, by part (vi) the final result is simply the union of the intervals.

Theorem 5.4 *The rules for precise widening are normalizing — the final set of intervals is independent of the order of application of the rules in the definition.*

Proof:

This observation follows from the use of down-sets of atoms in the Thm. 5.3 and the definition of the precise widening operator. Let V be the result of a precise narrowing. The down-set of atoms in V is exactly the union of the down-sets of atoms in each of the arguments. Assume there was another interval domain value V' such that $\downarrow V = \downarrow V'$. If the representation is different in V and V' then there must be some interval in $x \in V$ such that one of the following holds:

1. there exists $y \in V'$ such that $x \preccurlyeq y$ and $x \neq y$,

2. there exists $y, z \in V'$ such that $\{x\} \sqsubseteq \{y, z\}$,

If neither (1) nor (2) hold then there must be some element of x that is not covered by V', contradicting $\downarrow V = \downarrow V'$. If (1) holds with $x = [a..b]$ then either $a-1$ or $b+1$ must be in y and thus must also be in some interval of V. But such an interval would conjoin x meaning that V would not be a valid integer interval domain. Thus (1) cannot hold. If (2) holds, then by similar reasoning, y and z must be conjoint, implying that V' is not a valid integer interval domain. Thus V and V' must have the same representation.

□

The requirements for the relaxed widening operator are both more and less restrictive than those for the precise widening operator; less restrictive in terms of accuracy, but more restrictive in terms of convergence. We could simply choose to define the widening operator as returning $\top$. Although that satisfies the requirements due to the weak accuracy requirement, in practical terms such a definition would be nearly useless for discovering any information about expressions. On the other hand we obviously have to give away some of the accuracy in our domains in order to satisfy the convergence requirement. We deal with these somewhat contradictory demands by defining an operator that gives exact answers if there is a bound on the range of answers, yet converges very quickly if there is no bound. We use the precise operator to simplify the definition of the relaxed operator.

Defn 5.7 (Relaxed Integer Widening) *Given integer domains* $D_1 = \{V_1, V_2, \ldots, V_n\}$ *and* $D_2 = \{W_1, W_2, \ldots, W_m\}$, *with* $V_1 = [a_1..b_1], V_n = [c_1..d_1], W_1 = [a_2..b_2]$, *and* $W_m = [c_2..d_2]$,

$$(D_1 \nabla_R D_2) = D_1 \nabla_P D_2 \nabla_P \{U_1, U_2\}$$

where

$$U_1 = \begin{cases} [-\infty..a_2] & \text{if } a_2 < a_1 \\ \bot & \text{otherwise} \end{cases}$$

and

$$U_2 = \begin{cases} [d_2..\infty] & \text{if } d_2 > d_1 \\ \bot & \text{otherwise} \end{cases}$$

The basic intuition for the result of a relaxed widening operation is that if we have discovered bounds on potential results (represented with a domain value like

$\{[1],[50]\}$) then as long as further elements remain within the $\{[1],[50]\}$ range, we can maintain exact information without concern for divergence. If the range begins to "expand" towards either side of this range, we immediately expand the range in the direction of either infinity or negative infinity. This is obviously a fairly simple model, but it works surprisingly well in practice due to the nature of real code. Although this will be discussed in more detail in Section 6.2, a bit of intuition regarding the usefulness of this operator is in order at this point. If a code fragment has (and enforces!) upper and lower bounds for expected values, these bounds will be encoded in the program by way of conditional expressions that either provide default values if the bounds are exceeded or perform some sort of error handling routine. In either case, the "normal" computation will have the expected range encoded in the program. Due to the fact that we "split" scopes (again, see Section 6.2) based on conditional expressions, these encoded bounds will "narrow" an estimate towards these encoded bounds. If a program has no encoded bounds then either no known bounds exist or the implementation is faulty in terms of not dealing with exceptional circumstances. In either case, we clearly cannot make any assumptions about the potential domain other than that the domain could be infinite. It is possible to encode more complex infinite domains as part of a different abstraction; this will be discussed in Section 7.2.2. In addition, we will discuss how to extend the implied concept of "direction" or "dimension" to deal with non-linear data models for non-integer domains.

Theorem 5.5 *(∇_R is Relaxed)*

Relaxed integer widening (Defn. 5.7) satisfies the conditions for relaxed widening (Defn. 3.2).

Proof:

There are three conditions that must be satisfied for ∇_R to satisfy Defn. 3.2. First, if $V = V_1 \nabla_R V_2$ then $\downarrow V \supseteq (\downarrow V_1) \bigcup (\downarrow V_2)$. Second, $\forall x, y \in V : x \neq y \Longrightarrow x \| y$. Third, for any function f and value x_0, there exists a k such that $f(x_k) \sqsubseteq x_k$ where $x_i = x_{i-1} \nabla_R f(x_{i-1})$ for $i > 0$.

We will deal with the first two conditions by appealing to the corresponding proof for the precise widening operator. The first condition for relaxed widening is more flexible; we only need to show that elements are not lost. Since the result is the precise widening of the original domains plus some additional elements, and since the

precise widening operator does not lose elements, we can conclude that the relaxed operator does not lose elements. The second requirement for relaxed widening is satisfied using the same proof as for precise widening.

The last part of our proof obligation is to show that the convergence statement holds for this widening operator. Consider a particular value in the abstract domain, $V = \{v_1, v_2, \ldots, v_n\}$ and define *min* and *max* as follows:

$$min = \begin{cases} b & \text{if } v_1 = [-\infty..b] \\ a & \text{if } v_1 = [a..b] \end{cases}$$

and

$$max = \begin{cases} a & \text{if } v_n = [a..\infty] \\ b & \text{if } v_n = [a..b] \end{cases}$$

Let the number of *free atoms* of V be 0 if $V = \top$. Otherwise let the number of free atoms be the cardinality of the set of integers between *min* and *max* that are not in V plus one for each direction in V that is not extended to infinity. The intuition is that we count each integer that falls in the "gaps" of V as a free atom plus a special marker atom for each direction in V that has not been extended to infinity. Since we know that any particular interval domain value has a finite representation, the number of "gaps" must be finite, so the total number of free atoms must be finite.

Claim: Any widening of V either decreases the number of free atoms by at least one, or produces V.

Consider a particular widening of V by some other value W. By definition, if $V = \top$ then $V \nabla_R W = V$, satisfying the claim. If there exists an element in W either larger than the maximum element in V or smaller than the minimum value in V then by Defn. 5.7 one of the directions is extended to infinity. This eliminates one of the special marker free atoms and thus satisfies the claim. If the elements of W are between the minimum and maximum values of V then we perform a precise widening. By Defn. 5.6, if every interval in W is below some interval in V then the result is V, again satisfying the claim. The final case to consider is when there exists $w \in W$ such that for all $v \in V$, $w \not\preccurlyeq v$. Then, by definition of $\preccurlyeq$, there exists some x such that $x \in w$ and for all $v \in V$, $x \notin v$. Since the definition of precise widening guarantees that the result contains x and all values of V, the number of free atoms in the result must be less than the number of free atoms in V.

Let k' be the number of free atoms in x_0. Since the claim is satisfied, the number of free atoms in x_i where $x_i = x_{i-1} \nabla_R f(x_{i-1})$ must be strictly less than the number of free atoms in x_{i-1}. Thus there exists some value $k \leq k'$ such that $x_k = x_k \nabla_R f(x_k)$. Thus by definition of ∇_R , $f(x_k) \sqsubseteq x_k$, and the convergence requirement is satisfied.

□

Although our precise widening operator commutes, relaxed widening does not. The reason for this is that relaxed widening conservatively extends a domain in the *direction* in which the domain is growing. The direction of growth is expressed in the order in which widenings are performed. For example, using our previous example we see that

$$\{[-10..5], [7..11]\} \nabla_R \ \{[-11..-1], [1..4], [13..13]\} = \{[-\infty..5], [7..11], [13..\infty]\}$$

but

$$\{[-11..-1], [1..4], [13..13]\} \nabla_R \ \{[-10..5], [7..11]\} = \{[-11..5], [7..11], [13..13]\}.$$

The resulting domains correctly express the behaviour of the respective widenings since in the first case the "direction" of the domain growth is towards infinity in both directions while in the second case the second domain is contained within the range of the first estimate. When such containment occurs there is no possibility of infinite growth so we can generate a better estimate while maintaining safety. Finally, note that the second relaxed estimate generates exactly the same result as a precise widening.

There is one aspect of domain definition that we have ignored in presenting the integer interval domains: we have not presented any definitions for primitive operations over intervals. Although we are not going to give the details of the operations, it is important to note that such definitions are part of the overall definition that is used by the partial evaluation system. In the next section, when we define the structural abstract domain, we will present detailed definitions of the primitive operations for lists.

5.1.3 A Larger Example using the Integer Domain

In order to illustrate the operation of both the algorithm and the integer domain, we will consider a function that sums numbers in the range from `start` to `stop`.

```
(define (Sum start stop)
   (if (> start stop)
       0
       (+ start (Sum (+ 1 start) stop))
) )
```

In order to have a reasonable size example, we will skip most of the "uninteresting" steps in the derivation and will focus on the recursive evaluations of Sum. In the example, we will evaluate Sum from 1 to x where x is unknown (i.e. ⊤). We assume that we have an accurate *Split* function.

Given an evaluation `(Sum (+ 1 start) stop)`, we will have a trace step of the form:

$$\texttt{(Sum } x \ y\texttt{)}\ \delta(\mathit{start})\ \delta(\mathit{stop})\ \xi$$

where x is the value of `(+ 1 start)`, y is the value of `stop`, and the δ and ξ values are as given. In terms of the evaluation, this captures the state of e_1^α for each argument and the state of δ and ξ immediately following line 3.15(1) in which the actual parameter is evaluated.

Each nested evaluation of the body will be indented; since the re-evaluation of the entire expression with the new ξ environment (in line 3.15(12)) is strictly tail-recursive, we will not indent for this case. Since all but the initial call to Sum occur as a result of evaluating the body of Sum, after each completed recursive evaluation of Sum we will give the overall value for e^α in the form "$e^\alpha = 0 \nabla_P (z + y)$". This reflects the basic evaluation strategy for the body of Sum — the conditional expression will always be unknown, so the overall result will always be a precise widening of the values of each branch. The value of the first conditional branch is always zero and $(z + y)$ is the value of the second conditional branch where z is the value of `start` during the evaluation of the body and y is the result of the recursive evaluation. It is very important to note that $z = \delta(\mathit{start}) \nabla_R x$ since, as defined by line 3.15(6), the body is evaluated in the ρ' environment found by widening the old δ value by the new e_1^α value.

Finally, after giving the new e^{α} value, we present a trace line that gives the value for ξ' which determines whether e^{α} is the result or whether another evaluation is necessary.

A sequence of trace lines from a recursive evaluation might look like the following:

```
(Sum 3 [2..∞]) 2 [1..∞] ⊥
      (Sum [3..∞] [2..∞]) [2..∞] [1..∞] ⊥
e^α = 0 ∇_P ([2..∞] + ⊥) = 0
ξ' = ⊥ ∇_R 0 = 0
```

The two evaluated parameter values are given in the `(Sum 3 [2..∞])` fragment of the first line. In this example, it is not the case that both parameter values are below the respective values in δ (represented by the next two values in the trace line). Thus an evaluation of the body results. The evaluation of the body (eventually) yields another recursive evaluation of `Sum`.

In the recursive evaluation, the two evaluated parameter values are given in the `(Sum [3..∞] [2..∞])` fragment. In this case each parameter value is below the respective value in δ (the next two values in the trace line). This means that in the algorithm the value returned would be the value of ξ, which in this case is $\perp$.

The next trace line shows the computed value for the body of `Sum` for the first evaluation. Note that the [2..∞] value in the expression $([2..\infty] + \perp)$ results from the value bound to `start` during the evaluation of the body. This value was calculated from a relaxed widening of the old δ value (i.e. 2) by the e_1^{α} value (i.e. 3).

The third line computes the new ξ' value which is always the old ξ value widened by the computed e^{α} value. In this case, the old ξ value is $\perp$ and the e^{α} value is 0. Since $e^{\alpha} \not\sqsubseteq \xi$, we must re-evaluate the original expression with the new ξ'.

```
(Sum 1 ⊤) ⊥ ⊥ ⊥                                  (Note: d = false)
      (Sum 2 [1..∞]) ⊥ ⊥ ⊥                       (Note: d = true)
            (Sum 3 [2..∞]) 2 [1..∞] ⊥
                  (Sum [3..∞] [2..∞]) [2..∞] [1..∞] ⊥
            e^α = 0 ∇_P ([2..∞] + ⊥) = 0
            ξ' = ⊥ ∇_R 0 = 0
```

$$
\begin{array}{l}
\text{(Sum 3 [2..}\infty\text{])} \ \ 2 \ \ [1..\infty] \ \ 0 \\
\qquad \text{(Sum [3..}\infty\text{] [2..}\infty\text{])} \ \ [2..\infty] \ \ [1..\infty] \ \ 0 \\
\qquad e^{\alpha} = 0 \ \nabla_P \ \ ([2..\infty] + 0) = \{0, [2..\infty]\} \\
\qquad \xi' = 0 \ \nabla_R \ \ \{0, [2..\infty]\} = \{0, [2..\infty]\} \\
\text{(Sum 3 [2..}\infty\text{])} \ \ 2 \ \ [1..\infty] \ \ \{0, [2..\infty]\} \\
\qquad \text{(Sum [3..}\infty\text{] [2..}\infty\text{])} \ \ [2..\infty] \ \ [1..\infty] \ \ \{0, [2..\infty]\} \\
\qquad e^{\alpha} = 0 \ \nabla_P \ \ ([2..\infty] + \{0, [2..\infty]\}) = \{0, [2..\infty]\} \\
\qquad \xi' = \{0, [2..\infty]\} \nabla_R \ \ \{0, [2..\infty]\} = \{0, [2..\infty]\} \\
e^{\alpha} = 0 \ \nabla_P \ \ (2 + \{0, [2..\infty]\}) = \{0, 2, [4..\infty]\} \\
\xi' = \perp \nabla_R \ \ \{0, 2, [4..\infty]\} = \{0, 2, [4..\infty]\} \\
\text{(Sum 2 [1..}\infty\text{])} \ \ \perp \ \ \perp \ \ \{0, 2, [4..\infty]\} \\
\qquad \text{(Sum 3 [2..}\infty\text{])} \ \ 2 \ \ [1..\infty] \ \ \{0, 2, [4..\infty]\} \\
\qquad \qquad \text{(Sum [3..}\infty\text{] [2..}\infty\text{])} \ \ [2..\infty] \ \ [1..\infty] \ \ \{0, 2, [4..\infty]\} \\
\qquad \qquad e^{\alpha} = 0 \ \nabla_P \ \ ([2..\infty] + \{0, 2, [4..\infty]\}) = \{0, [2..\infty]\} \\
\qquad \qquad \xi' = \{0, 2, [4..\infty]\} \nabla_R \ \ \{0, [2..\infty]\} = \{0, [2..\infty]\} \\
\qquad \text{(Sum 3 [2..}\infty\text{])} \ \ 2 \ \ [1..\infty] \ \ \{0, [2..\infty]\} \\
\qquad \qquad \text{(Sum [3..}\infty\text{] [2..}\infty\text{])} \ \ [2..\infty] \ \ [1..\infty] \ \ \{0, [2..\infty]\} \\
\qquad \qquad e^{\alpha} = 0 \ \nabla_P \ \ ([2..\infty] + \{0, [2..\infty]\}) = \{0, [2..\infty]\} \\
\qquad \qquad \xi' = \{0, [2..\infty]\} \nabla_R \ \ \{0, [2..\infty]\} = \{0, [2..\infty]\} \\
\qquad e^{\alpha} = 0 \ \nabla_P \ \ (2 + \{0, [2..\infty]\}) = \{0, 2, [4..\infty]\} \\
\qquad \xi' = \{0, 2, [4..\infty]\} \nabla_R \ \ \{0, 2, [4..\infty]\} \{0, 2, [4..\infty]\} \\
0 \ \nabla_P \ \ (1 + \{0, 2, [4..\infty]\} \\
\{0, 1, 3, [5..\infty]\}
\end{array}
$$

The residual that we would produce is as follows:

```
((lambda (stop)
   (if (> 1 stop)
       0
       (+ 1 ((lambda (stop)
               (if (> 2 stop)
                   0
                   (+ 2 (Sum 3 stop))))
             stop)
  )   )   )
 x)
```

The basic intuition about the structure of the residual is that the known constant values of `start` are inlined and the parameter is removed. During the evaluation, once `start` becomes an abstract value that cannot be concretized, then we revert to the general function call. In terms of the trace, the final result and the last two e^{α} computations are the evaluations that actually create the residual. Note that the for the call to `Sum` in the residual is slightly different than what the formal algorithm would produce. Line 3.15(3) substitutes the body of `Sum` rather than just its function identifier. The formal algorithm avoids dealing with function identifiers in order to reduce the complexity of the algorithm; the substitution is trivial to make in the given residual.

5.2 Structured Domains

As with the integer abstract domain, we begin by defining an abstract domain for structured values. In keeping with the basic Scheme flavour of our language, we will adopt Scheme's S-expression model for structured domains. Each value in the domain is either an *atom* or a pair of values. Atoms are non-structural values; for our purposes we will assume that atoms are either integers or the special value `NIL`. The basic list operators are pair construction (`cons`), extraction of the first value of a pair (`car`), and extraction of the second value of a pair (`cdr`). List predicates will be restricted to `null?` and `atom?`; it is a straightforward exercise to build predicates

such as `list?`. We will assume the simple list model without imperative operators such as `set-car!` or `set-cdr!`.

Due to the requirements for the precise widening operator, given two abstract structure values, we need to be able to represent exactly the information in the two representations. The basic approach that we will adopt is to keep sequences of lists. The precise widening operator will then simply involve adding another list to the sequence. The relaxed widening operator that we will define preserves guaranteed structure and value estimates, but performs substantial simplifications. The basic approach for the relaxed operator is to merge *all* of the lists into a single list where the single list preserves as much structure as possible about the original list.

Defn 5.8 (List Domain)
A value in a list domain, $D^L = \{d_1, \ldots, d_n\}$, *is a sequence of lists.*

Defn 5.9 (Precise List Widening) *Given list domain values D_1 and D_2, we define $D_1 \nabla_P D_2$ to be $D_1 @ D_2$ where "@" is a sequence concatenation operator.*

We will let "@" remain as an informal operation for now; after we define the ordering relationship between domain values we will more carefully define the meaning of concatenation. For the time being, simply assume that concatenation does not admit "redundant" lists into the sequence.

As soon as we admit Scheme style lists into our system, we allow heterogeneous types which, in an language without a compile-time type system, necessitates the use of some sort of type identification. We will use $\tau(x)$ to denote the type of x; the universe of types for our system as defined so far is $\{Integer, List\}$ where `NIL` is considered to be a list, i.e. $\tau(\texttt{NIL}) = List$. We recursively define the *merge* of two lists

as follows:

$$merge(x,y) = \begin{cases} \top & \text{if } x = \top \text{ or } y = \top \\ \top & \text{if } \tau(x) \neq \tau(y) \\ x \ \nabla_R^{\tau(x)} \ y & \text{if } \tau(x) \neq \mathit{List} \text{ and } \tau(x) = \tau(y) \\ \texttt{NIL} & \text{if both of } x, y \text{ are } \texttt{NIL} \\ \top & \text{if only one of } x, y \text{ are } \texttt{NIL} \\ \texttt{(cons } merge\texttt{((car } x\texttt{) (car } y\texttt{))} & \\ \qquad merge\texttt{((cdr } x\texttt{) (cdr } y\texttt{)))} & \text{otherwise} \end{cases}$$

We extend the notation for *merge* by defining $merge(x_1, x_2, \ldots)$ as being equivalent to $merge(\ldots(merge(x_1, x_2), x_3), \ldots)$. In addition, given a domain value $X = \{x_1, \ldots\}$, we define $merge(X) = merge(x_1, \ldots)$ where $merge(X) = X$ if there is only one list within X. Note that due to the relaxed widening operation performed by merge, a merge operation is not necessarily an associative operation.

> **Defn 5.10 (Relaxed List Widening)** *Given list domain values* $X = \{x_1, x_2, \ldots\}$ *and* $Y = \{y_1, y_2, \ldots\}$, *we define* $X \ \nabla_R \ Y$ *to be a list* $V = merge(v_1, v_2, \ldots)$ *where* $v_i = merge(x_i, Y)$

We generally follow Scheme syntax for lists: `(1 2 3)` represents the construction `(cons 1 (cons 2 (cons 3 NIL)))`. If the list does not end with `NIL`, the list is represented with a dot between the last pair of elements. For example, `(1 2.3)` represents the construction `(cons 1 (cons 2 3))`. We will generally not put the Scheme backquote on our lists unless necessary to clarify the meaning. The following are a few examples of relaxed widening operations using this abstract domain.

$$\texttt{NIL} \ \nabla_R \ \texttt{(1 2)} = \top$$
$$\texttt{(1 2 3)} \nabla_R \ \texttt{(1 2 T)} = \texttt{(1 2 T)}$$
$$\texttt{(1 2 3 4 5)} \nabla_R \ \texttt{(1 2 T)} = \texttt{(1 2 T.T)}$$
$$\texttt{(1 2 3.T)} \nabla_R \ \texttt{(1 3 T)} = \texttt{(1 [2..}\infty\texttt{] T.T)}$$
$$\texttt{(1 (2) 3)} \nabla_R \ \texttt{(1 2)} = \texttt{(1 T.T)}$$
$$\texttt{(1 (2) 5)} \nabla_R \ \texttt{(1 (3 4) 5)} = \texttt{(1 ([2..}\infty\texttt{].T) 5)}$$

These examples illustrate the structure preserving nature of the relaxed widening operator. If the static knowledge about the structures is consistent, we are able to preserve the structural information, even if the particular elements become (fully or partially) dynamic. The first example demonstrates a complete loss of structural information, while the second example maintains complete structural information though it loses some information about elements. The third example is interesting in that there is only a partial loss information about both the elements and the structure of the original lists. Note that we do know that we have *at least* three elements even though we do not know the value of the third element. The fourth example contains the result of a non-trivial integer relaxed widening operation. In the fifth example we lose all information about the nested list, but retain the structural information about the outermost list. In the last example we lose partial information about the nested structure while retaining the complete structure of the outermost list.

The basic idea of relaxed widening is first to capture the direction of growth in the abstract domain and then to compress all of the lists into a single list. It is important to note here that there is no concept of "direction" in the list domain itself; the reason that there is any concern about direction of growth is that the *merge* operation could apply relaxed widening operators from other domains and these operators may have some idea of direction of growth. In general this operation could lose a great deal of accuracy in directional domains such as the integer intervals due to the double merge. We could avoid this loss of accuracy by not performing the second phase of the merging (i.e. by letting the result be the sequence $v_1, v_2, \ldots$ rather than the merge of these lists), but performing the second phase of merging provides a more efficient version of the list representations that is still a safe approximation. Defining relaxed widening such that reasonably compact notations result makes the evaluation of dynamic recursions much more efficient. Thus we only pay the cost of manipulating potentially large sequences of lists only when we care about having very accurate results, namely when we are performing a static evaluation. Although we do perform precise widening operations within a dynamic evaluation, due to the fact that relaxed widening operations occur on all parameters for each dynamic call, there are a bounded number of precise widenings before a relaxed widening operation occurs.

The next step is to define the meaning of the abstract primitive list operations and predicates. We will use *cons*, *car*, and *cdr* to represent the abstract versions of the primitive operators and *atom*? and *null*? to represent the abstract versions of the

predicates. Let $X = \{x_1, \ldots\}$, and $Y = \{y_1, \ldots\}$ be abstract list domain values.

$$(\textit{atom?}\ \ X) = \begin{cases} \texttt{true} & \text{if } \forall x_i, x_i = \texttt{NIL} \text{ or } \tau(x_i) = \textit{Integer} \\ \texttt{false} & \text{if } \forall x_i, x_i = (\texttt{cons}\ \ v\ \ L) \\ \top & \text{otherwise} \end{cases}$$

$$(\textit{null?}\ \ X) = \begin{cases} \texttt{true} & \text{if } \forall x_i, x_i = \texttt{NIL} \\ \texttt{false} & \text{if } \forall x_i, x_i = (\texttt{cons}\ \ v\ \ L) \\ \top & \text{otherwise} \end{cases}$$

$$(\textit{car}\ \ X) = \{v_1, v_2, \ldots\} \text{ where } \ v_i = \begin{cases} \bot & \text{if } \tau(x_i) \neq \textit{List} \text{ or } x_i = \texttt{NIL} \\ \top & \text{if } x_i = \top \\ v & \text{if } x_i = (\texttt{cons}\ \ v\ \ L) \end{cases}$$

$$(\textit{cdr}\ \ X) = \{v_1, v_2, \ldots\} \text{ where } \ v_i = \begin{cases} \bot & \text{if } \tau(x_i) \neq \textit{List} \text{ or } x_i = \texttt{NIL} \\ \top & \text{if } x_i = \top \\ L & \text{if } x_i = (\texttt{cons}\ \ v\ \ L) \end{cases}$$

$$(\textit{cons}\ X\ Y) = \{v_1, v_2, \ldots\} \text{ where } \ v_i = (\texttt{cons}\ \ \textit{merge}(X)\ \ y_i)$$

The most interesting of these definitions is the *cons* definition. Recall that $\textit{merge}(X)$ is defined to be the *merge* of the lists of X. The intuition for the *cons* rule is that we first normalize the element that we are about to cons onto the lists in Y and then create the new sequence of lists by adding this normalized element onto each list in Y.

Although we have now defined the domains and operations, we still have to insure that these definitions are safe to use within our framework. In order to make such a claim, we must define the $\preccurlyeq$ ordering relation over lists, show that the widening operators satisfy their respective constraints, and show that the operators are safe.

5.2.1 Analysis of the Abstract Structural Domain

Given two lists, x and y, we say $x \preccurlyeq y$ if one of the following holds:

$$y = \top \tag{5.20}$$

$$x = \texttt{NIL} \text{ and } \tau(y) = \mathit{List} \tag{5.21}$$

$$\tau(x) = \tau(y) \text{ and } \tau(x) = \mathit{Integer} \text{ and } x \sqsubseteq^{\tau(x)} y \tag{5.22}$$

$$x = (\texttt{cons } a\ b) \text{ and } y = (\texttt{cons } c\ d) \text{ and } a \preccurlyeq c \text{ and } b \preccurlyeq d \tag{5.23}$$

Note that we use $x \sqsubseteq^{\tau(x)} y$ within our definition of $\preccurlyeq$ for lists; this is due to the fact that lists are heterogeneous and we would like to be able to retain as much accuracy as possible about the elements within the list. This aspect of the definition corresponds to the $\nabla_R^{\tau(x)}$ case for integer x in the definition for *merge* given earlier.

There is a reasonably intuitive characterization for this ordering relationship — list x is below list y if x is provably longer than y or if every element in x is below the corresponding element in y. There are several interesting incomparability aspects to this ordering relationship. First, any integer value is incomparable to any list value (including NIL). This corresponds to one's intuition that elements of different types cannot be ordered with respect to each other unless there are explicit coercion operators which our language does not support. Second, lists of the same length that have $\top$ elements in different locations are incomparable. Finally, since the definition is recursive, nested lists fit naturally into the relationship. The following are a few examples of valid $\preccurlyeq$ relationships:

$$\begin{aligned}
(1\ 2) &\preccurlyeq \top \\
(1\ 2\ 3) &\preccurlyeq (1\ 2\ \top) \\
(1\ 2\ 3\ 4\ 5) &\preccurlyeq (1\ 2\ \top.\top) \\
(1\ 2) &\preccurlyeq (1\ 2\ \top.\top) \\
(1\ 2\ 3.\top) &\preccurlyeq (1\ [2..\infty]\ \top.\top) \\
(1\ (3\ 4)\ 5) &\preccurlyeq (1\ ([2..\infty].\top)\ 5)
\end{aligned}$$

The first example follows directly from condition 5.20. The second example uses $\sqsubseteq^I$ from condition 5.22 to verify the below relationship for the first two list elements; $\preccurlyeq$ holds for the third element by condition 5.20. Examples three and four are interesting in that we have what appears to be both a longer and a shorter list being below

the list `(1 2 T.T)`. The shorter list is below `(1 2 T.T)` since NIL is below the cons cell `(cons T.T)`. When the longer list is compared to `(1 2 T.T)`, the interesting comparison is when we compare `(cons 3 (cons 4...))` to `(cons T T)`. Since both $3 \preccurlyeq \top$ and `(cons 4...)` $\preccurlyeq \top$ by 5.20, the relationship holds. The basic observation is that if the structure of a list is not completely known, then the last cons cell will be of the form `(cons` x `T)` for some x.

Nearly all of these examples were taken from the earlier examples for the widening operator. Recall that the ∇_R operation must guarantee that for any x and y, the result of $x \nabla_R y$ must be above either x of y. By inspection, this relationship holds for the examples we have given; we still must prove that the ∇_P and ∇_R operators are in fact valid precise and relaxed widening operators with respect to the list domain.

Before addressing the validity of the ∇_P and ∇_R operations, we need to deal with two other issues: first, we need to clarify the operation of the list concatenation operator "@", and second, we need to verify that the $\sqsubseteq$ operation holds for sequences of lists.

When we defined ∇_P for the list domain, we described "@" as simple sequence concatenation. In reality, the concatenation operator will only add new lists onto the end of the sequence when no redundant information will appear in the resulting sequence. More formally, given sequences $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots\}$ we recursively define sequence concatenation as follows:

$$X @ Y = \begin{cases} X & \text{if } Y = \{\} \\ X @ \{y_2, \ldots\} & \text{if there exists } x_i \text{ such that, } y_1 \sqsubseteq x_i \\ \{x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n\} @ Y & \text{if } x_i \sqsubseteq y_1 \\ \{x_1, \ldots, x_n, y_1\} @ \{y_2, \ldots\} & \text{if for all } x_i,\ y_1 \| x_i \end{cases}$$

The first case is the trivial identity operation. The second case ignores a particular list in Y if the information in that list is already represented in X. The third case is symmetric with the second but for lists in X that are below a list in Y, and the final case actually adds a list to a sequence since the added list represents different (incomparable) information.

The next step is to discuss the $\sqsubseteq$ relation. Recall that the definition of $\sqsubseteq$ is that given abstract domain values x and y, we will say that $x \sqsubseteq y$ if $\downarrow x \subseteq \downarrow y$ where $\downarrow x$ is the set of *atoms* below x. Note that in this context we mean *atom* in the lattice sense,

not in the Scheme list sense. In this section, we will continue to use the font "*atom*" to mean a lattice *atom* rather than the font "atom" to mean Scheme atom.

We define an *atom* in the list domain to be a list in which no list element has the value $\top$ and in which all components of the list are *atoms* in their respective domains. Thus `(1 2 3)` and `(1 (2 3))` are *atoms* but `(1 (2.⊤) 4)` and `(1 [2..∞] 4)` are not. The latter two lists are not *atomic* since in the first case $\top$ appears within the list, while in the second case a non-*atomic* integer domain value appears — the interval `[2..∞]`. A list such as `(1 [2..∞] 4)` is above every list of the form `(1` x `4)` where x is any integer domain value below `[2..∞]`.

Theorem 5.6 *(∇_P^L is Precise)*

Precise list widening (Defn. 5.9) satisfies the conditions for precise widening (Defn. 3.1).

Proof:

There are two conditions that must be satisfied for ∇_P to satisfy Defn. 3.1. First, if $V = X \nabla_P Y$ then $\downarrow V = (\downarrow X) \bigcup (\downarrow Y)$. Second $\forall x, y \in V : x \neq y \Longrightarrow x \| y$.

We will first deal with the incomparability requirement. By definition of the "@" operator, the only time that a list is added to a sequence is when the new list is incomparable to all existing lists. Thus the incomparability requirement must be satisfied.

The second requirement is that the down-set of the result of a precise widening is equal to the union of the down-sets of the of the two original abstract values. We prove this by showing that each side is a subset of the other.

Part 1: $\downarrow V \subseteq (\downarrow X) \bigcup (\downarrow Y)$. Let v be some *atom* in $\downarrow V$. Then, by definition of $\downarrow$, there exists some list $v' \in V$ such that $v \preccurlyeq v'$. By case analysis of "@", any list in V exists in at least one of X or Y. Thus v' is in at least one of X or Y and $v \in (\downarrow X) \bigcup (\downarrow Y)$.

Part 2: $\downarrow V \supseteq (\downarrow X) \bigcup (\downarrow Y)$. Let x be some *atom* in $(\downarrow X) \bigcup (\downarrow Y)$. Then, by definition of $\downarrow$ and $\bigcup$, there exists some list x' in at least one of X or Y such that $x \preccurlyeq x'$. Assume $x' \in X$. By case analysis of "@" either $x' \in V$ or there exists some $y' \succcurlyeq x'$ such that $y' \in V$. But then by transitivity, $x \preccurlyeq x' \preccurlyeq y' \Longrightarrow x \preccurlyeq y'$. Thus since $y' \in V$ we conclude that $x \in \downarrow V$.

□

We must now show that the relaxed widening operator is correct. The properties for the list domain relaxed widening are dependent on the definition of *merge* so we will first prove that *merge* generates safe results.

Theorem 5.7

Given elements x and y, $merge(x, y) \succcurlyeq x$ and $merge(x, y) \succcurlyeq y$.

Proof:

The proof is by induction over the structure of x and y. Note that we only use *merge* inductively in the final case when both x and y are cons cells; the other cases cover all other base conditions.

Case 1: $x = \top$ or $y = \top$. Then $merge(x, y) = \top$. By definition, $x, y \preccurlyeq \top$ so $merge(x, y) \succcurlyeq x, y$.

Case 2: $\tau(x) \neq \tau(y)$. Then $merge(x, y) = \top$. By definition, $x, y \preccurlyeq \top$ so $merge(x, y) \succcurlyeq x, y$.

Case 3: $\tau(x) \neq List$ and $\tau(x) = \tau(y)$. Then $merge(x, y) = x \ \nabla_R^{\tau(x)} \ y$. By definition of relaxed widening, $x, y \preccurlyeq x \ \nabla_R \ y$ so $merge(x, y) \succcurlyeq x, y$.

Case 4: $x, y = \texttt{NIL}$. Then $merge(x, y) = \texttt{NIL}$, so trivially, $merge(x, y) \succcurlyeq x, y$.

Case 5: $x = \texttt{NIL}, y \neq \texttt{NIL}$. Then $merge(x, y) = \top$, so trivially, $merge(x, y) \succcurlyeq x, y$.

Case 6: $x \neq \texttt{NIL}, y = \texttt{NIL}$. Then $merge(x, y) = \top$, so trivially, $merge(x, y) \succcurlyeq x, y$.

Case 7: $x =$ `(cons a b)` and $y =$ `(cons c d)`. Then $merge(x, y)$ is defined to be (cons *merge*((car x) (car y)) *merge*((cdr x) (cdr y))). By definition of `car` and `cdr`, this is equivalent to (cons *merge*(a, c) *merge*(b, d)). By induction we assume that $merge(a, c) \succcurlyeq a, c$ and $merge(b, d) \succcurlyeq b, d$ so by definition of $\succcurlyeq$, we can conclude that $merge(x, y) \succcurlyeq x, y$.

□

Theorem 5.8 *(∇_R^L is Relaxed)*

Relaxed list widening (Defn. 5.10) satisfies the conditions for relaxed widening (Defn. 3.2).

Proof:

There are three conditions that must be satisfied for ∇_R to satisfy Defn. 3.2. First, if $V = X \nabla_R Y$ then $\downarrow V \supseteq (\downarrow X) \bigcup (\downarrow Y)$. Second, $\forall x, y \in V : x \neq y \Longrightarrow x \| y$. Third, for any function f and value x_0, there exists a k such that $f(x_k) \sqsubseteq x_k$ where $x_i = x_{i-1} \nabla_R f(x_{i-1})$ for $i > 0$.

By construction, the result of ∇_R is a sequence containing a single list that results from a series of *merge* operations. By Thm. 5.7, given elements x and y, $x \preccurlyeq merge(x, y)$ and $y \preccurlyeq merge(x, y)$. By transitivity of $\preccurlyeq$, for any x_i, $x_i \preccurlyeq merge(\ldots, x_i, \ldots)$. This implies that for all $x_i \in X$, $x_i \preccurlyeq X \nabla_R Y$ and for all $y_i \in Y$, $y_i \preccurlyeq X \nabla_R Y$. Thus, by the definition of $\sqsubseteq$, $X \sqsubseteq X \nabla_R Y$ and $Y \sqsubseteq X \nabla_R Y$. This implies that $\downarrow V \supseteq (\downarrow X)$ and $\downarrow V \supseteq (\downarrow Y)$. Thus $\downarrow V \supseteq (\downarrow X) \bigcup (\downarrow Y)$.

The second condition for relaxed widening is trivially true since the relaxed widening in the list abstract domain returns a sequence containing a single list.

The final condition requires that any sequence of widening operations converges. Let $d(x)$ be the distance of x from $\top$ with respect to some function f. We assume that all non-list relaxed widening operators are valid. If $\tau(x) \neq List$, then let x^k be the bound for the number of widening operations using $\nabla_R^{\tau(x)}$ before $\nabla_R^{\tau(x)}$ converges with respect to function f. We then define $d(x)$ as follows:

$$d(x) = \begin{cases} 0 & \text{if } x = \top \\ 1 & \text{if } x = \texttt{NIL} \\ x^k & \text{if } \tau(x) \neq List \\ 1 + d(\texttt{(car } x\texttt{)}) + d(\texttt{(cdr } x\texttt{)}) & \text{otherwise} \end{cases}$$

Observation: If $x \neq \top$ then $d(x) > 0$.

Since each widening operation is simply a sequence of *merge* operations, it is sufficient to show that each $merge(x, y)$ operation is either an identity operation for x or that $d(merge(x, y))$ is strictly less than $d(x)$.

The proof is by induction over the structure of x and y. Note that we only use *merge* inductively in the final case when both x and y are cons cells; the other cases cover all other base conditions.

Case 1: $x = \top$ or $y = \top$. If $x = \top$ then $merge(x, y) = \top$ and $merge(x, y) = x$. If $x \neq \top$ then $d(merge(x, y)) = 0$ and $d(x) > 0$.

Case 2: $\tau(x) \neq \tau(y)$. Then $merge(x, y) = \top$ and $d(merge(x, y)) = 0$. Since $x \neq \top$, $d(x) > 0$.

Case 3: $\tau(x) \neq List$ and $\tau(x) = \tau(y)$. Then $merge(x, y) = x \ \nabla_R^{\tau(x)} \ y$. By definition of $d(x)$ and $\nabla_R^{\tau(x)}$, either $d(x \ \nabla_R^{\tau(x)} \ y) = d(x) - 1$ or $x = x \ \nabla_R^{\tau(x)} \ y$. Thus either $merge(x, y) = x$ or $d(merge(x, y)) < d(x)$.

Case 4: $x, y = \texttt{NIL}$. Then $merge(x, y) = \texttt{NIL}$, so trivially, $merge(x, y) = x$.

Case 5: $x = \texttt{NIL}, y \neq \texttt{NIL}$. Then $merge(x, y) = \top$, so trivially, $d(merge(x, y)) < d(x)$.

Case 6: $x \neq \texttt{NIL}, y = \texttt{NIL}$. Then $merge(x, y) = \top$, so trivially, $d(merge(x, y)) < d(x)$.

Case 7: $x =$ `(cons a b)` and $y =$ `(cons c d)`. Then $merge(x, y)$ is defined to be (cons $merge$((car x) (car y)) $merge$((cdr x) (cdr y))). By definition of `car` and `cdr`, this is equivalent to (cons $merge(a, c)$ $merge(b, d)$). By induction we assume that $merge(a, c) = a$ or $d(merge(a, c)) < d(a)$ and that $merge(b, d) = b$ or $d(merge(b, d)) < d(b)$. If $merge(a, c) = a$ and $merge(b, d) = b$ then $merge(x, y) = x$. In each of the other three cases, $d(merge(x, y)) < d(x)$.

Finally, since for any function f and value x_0, there is some k such that $d(x_0) = k$, we know that any series of *merge* operations will converge after no more than k merges. Thus ∇_R converges.

□

The basic intuition behind the convergence condition is that the list that results from $merge(x, y)$ will never be a longer list than either of x or y. If the resulting list is structurally the same as x and y then it will either be identical to x or there will be some value in the list that has moved closer to its fixed point. There is a measure of asymmetry in these statements — our definition for *merge* (and thus for ∇_R) is in fact associative with respect to *structure* but since widening operators in other domains, such as the integer interval domain, may not be associative, the overall statement of convergence can not take advantage of the structural associativity.

5.2.2 On the Expressiveness of the *List* Abstract Domain

Although the *List* abstract domain is a very simple model, it is surprisingly expressive when compared to other approaches. In particular, we will compare this approach to the accepted "state of the art" in off-line structural BTA — Launchbury's

uniform projections approach [54]. The basic intuition for Launchbury's approach is to create static and dynamic projections of programs; each projection encapsulates the respective aspects of the original program.

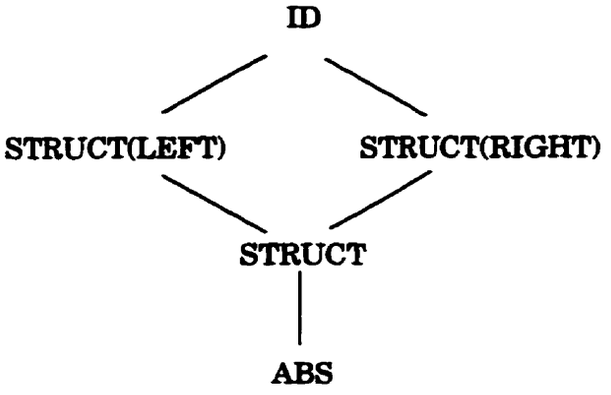


Figure 5.2.1: BTA Lattice for Structural Projection

One of the standard examples that Launchbury discusses is an association list program. Translated to a Scheme-like syntax, the program is as follows:

```
(define lookup (lambda (list value)
       (if (null? list)
           fail
           (if (equal? (car list) value)
               value
               (lookup (cdr list) value))
))     )
```

When the uniform projection approach is applied, the possible binding time annotations are the values in the lattice shown in Figure 5.2.1. The ABS value in the lattice means that the entire structure is "abstract", or dynamic. The STRUCT values means that the structure is known but that neither the left or right components of the list elements are known. The STRUCT(LEFT) and STRUCT(RIGHT) values mean that the structure is known and that, respectively, the left or right component of each element is known. The ID value means that the entire structure is known.

There are a few important differences in expressivity between this model and the abstract *List* domain that we have defined. First, the projection model is a *uniform*

model which means that if any list element is given a particular annotation, then *all* succeeding elements must have the same annotation. Such a model is not able to express changes in value annotations throughout the list. For example, given a list such as `(("a" 1) ("b" 2) ("c" T))`, the annotation for the list would be `STRUCT(LEFT)`, even though only a single value in the list is unknown. The second issue is that the effective model that a projections approach builds is based on knowledge about fixed structural components. This structural knowledge is discoverable in Launchbury's work because the source language is a derivative of ML [64] and as such, has explicit constructors used to build structures. Structures themselves in ML are structurally uniform, making it reasonable to apply a projections based approach. It is less clear how accurate a projections-based model could be in a more heterogeneous language such as Scheme. Additionally, languages such as C, in which side-effects are common, would not lend themselves to this type of model since a single assignment to an otherwise fully static structure would cause the loss of a great deal of static information. Part of the problem is inherent to using off-line approaches, but requiring full uniformity is likely to cause over-generalization in many situations.

In comparison, the on-line approach with the proposed *List* abstract domain is both a simpler model and is able to exploit additional static information. For example, consider the association list `lookup` function with an association list of `(("a" 1) (T 2) ("c" T))` and the requests `(lookup list "a")`, `(lookup list "b")`, `(lookup list "c")`. Using projections, none of these requests would be specialized; all searches would occur at run-time. Adopting the on-line approach with the *List* abstract model, the residual for the first search would be the constant `1`. The `lookup` residual for the second search would be as follows:

```
(lambda (list)
     (  (lambda (list)
          (if (equal? (car list) "b") 2 fail
         )  )
        (cdr list)
)    )
```

The outer `lambda` is the residual for checking element `"a"` — notice that this has removed the check for element `"a"`, leaving only the call for checking the rest of the list (i.e. the `(cdr list)`). The inner `lambda` is the residual for checking if the association list name for this element is `"b"`. If there is a match, we return the

inlined constant 2, otherwise we check the rest of the list. The result of "checking the rest of the list" is *fail* since the system is able to guarantee that `"b"` does not occur in the rest of the list.

Now consider the final example — `(lookup list "c")`. In this example, the system makes the same decisions regarding the first two elements in the association list, but is able to determine that the result of searching the rest of the list is known to produce the constant 3 rather than *fail*. The residual is as follows:

```
(lambda (list)
   (
      (lambda (list)
         (if (equal? (car list) "b")
             2
             3
          )   )
          (cdr list)
) )
```

If the association list insertion routine guaranteed unique instances of identifiers in the list, we would like to have the check of the second element removed and simply generate the constant 3. However, unless there were explicit uniqueness constraints provided to the interpreter, such a result is unlikely to be found by any system.

It is possible for a residual to contain a general call to the original `lookup` function, but this only occurs when the remaining part of the list is completely unknown. This is the point at which the *List* abstraction follows the same generalization as the uniform projections approach — once a particular cons cell has $\top$ in its cdr, we uniformly model the rest of the list as fully dynamic. Although it may be possible to develop consistent models that are non-uniform in this regard, that is a topic for future research.

Chapter 6

Implementation Issues

6.1 Design Overview

6.1.1 The Language

We built a prototype implementation of our system for a small subset of Scheme [19]. Scheme is an untyped functional programming language similar to Lisp [78]. The subset of the language that we model includes global and local scoping, `let` bindings, function definition (using either `define` or `lambda` style definitions), anonymous functions, list support (`cons`, `car`, `cdr`, etc.), and the imperative features `set!`, `set-car!`, and `set-cdr!`. We do not deal with features such as arrays, association lists, macros, and iterative control flow. The omitted features do not introduce new conceptual problems, and were omitted due to time constraints.

Our interpreter is a Scheme-to-Scheme transformation system. Due to the nature of the system we were building, we did make one significant change to the normal semantics of Scheme programs. Normally, when a Scheme interpreter evaluates an undefined variable, the interpreter generates an error message and terminates the calculation whereas in our interpreter, any undefined variable is considered as having the unknown value, `⊤`. This change allows for completely automatic evaluation of expressions within the interpreter. For example, assume that the main driver for a Scheme program is a function called `Main`. Further, assume that `Main` takes as its argument a file-stream value from which it does input and output. To partially evaluate the entire program, `Main` is simply applied to a variable that is not bound in the

global scope. The partial evaluator then interprets the entire program without knowing anything about the run-time input to the program and produces an appropriate residual.

This approach does have some implications about the state of the world at the time that the partial evaluation is performed. Any state in the interpreter that exists when the partial evaluation begins could be incorporated in the residual that is produced. This could be dangerous in general but is easy to fix by providing a "compiler" style interface to the evaluator that ensures that all initial run-time state variables (such as default file-stream variables) are uninitialized before beginning the interpretation. We have not yet implemented such an interface to the system.

The implementation itself was written in ML [64] using Standard ML of New Jersey (SML-NJ) Version 0.93. This choice was made early in the system development and allowed for an implementation decomposition that corresponds to the abstract decomposition presented in earlier chapters. This choice also incurred performance penalties that could possibly have been alleviated by adopting the CAML implementation of ML, but as the implementation was a proof-of-concept prototype only, the performance issue was not deemed to be worth the effort needed to port the system.

6.1.2 Structural Decomposition

The implementation separates the details of the abstract domains and the actual interpretation algorithm. For each natural concrete domain the system requires the definition of a corresponding abstract domain. Each domain in the system is built from an ML `structure`; the interpreter provides a set of required `signatures`, and `functors` are used to compose appropriate structures. In terms of other languages, `structures` (roughly) correspond to *packages* or *modules* in Ada [80] or Modula-3 [37] while `functors` (roughly) correspond to *generics* in those languages or *templates* in C++ [79]. An ML `signature` provides the interface requirements that a `structure` is required to satisfy; the polymorphic types in ML allow these signatures to be very general.

S-expressions are the fundamental structure describing entities in a Scheme program. An S-expression is composed of either an *atom* or a *pair* of S-expressions. In our system, the values of S-expressions are composed of abstract values; see Figure 6.1.1 for a diagram of the basic design of the interface between the interpreter

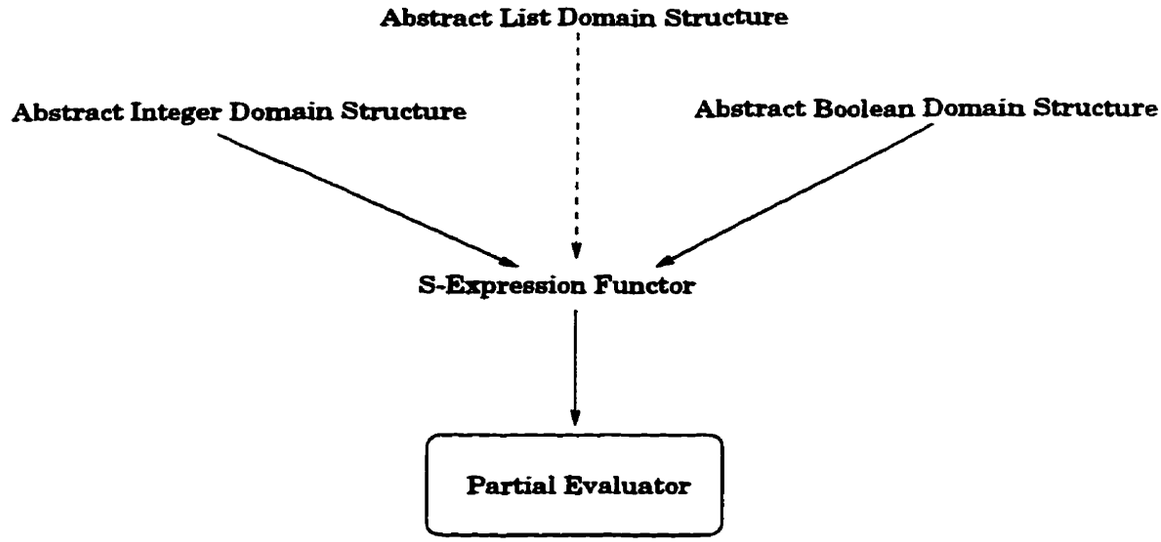


Figure 6.1.1: Implementation Structure

and the abstract domain definitions. The S-expression functor defines the main S-expression datatype in terms of the abstract definitions defined by the given abstract structures where each of the abstract structures must match a corresponding signature defined by the interpreter. The signatures for the abstract domains are defined in terms of the lifted and topped natural concrete domains as described in Section 3.3.1. All domain constraints are defined in terms of a global datatype named `lattice`. The `lattice` datatype is defined as:

```
datatype α lattice = TOP | BOT | ELEM of α
```

where α is typed as " `'a`" in the actual code. This type means that we can create lattices from other types; the constructor `ELEM` takes a value of any type and returns an element from the corresponding lattice type. This constructor does not guarantee formal lattice behaviour of the resulting elements; it is the programmers responsibility to ensure that the induced α `lattice` datatype is in fact a lattice.

Before discussing more details of the structure, we should clarify the status of the abstract list domain structure in our implementation. In the prototype, we integrated the abstract list operations directly into the interpreter. This was primarily due to the manner in which the code evolved and does not imply that there are fundamental difficulties in the conceptual decomposition. The dashed line in the arrow from this abstract domain to the interpreter indicates that this is a conceptual organization,

not an actual organization in the code.

Interfaces to Abstract Domains

Given the basic lattice datatype, we can define the interfaces for our abstract types. The `boolean` abstract domain signature is defined as:

```
signature BOOLBASETYPE = sig
   type domaintype;
   val alpha  : bool lattice -> domaintype lattice;
   val gamma  : domaintype lattice -> bool lattice;
   val equal  : domaintype lattice * domaintype lattice
                  -> bool lattice;
   val pwided : domaintype lattice * domaintype lattice
                  -> domaintype lattice;
   val rwided : domaintype lattice * domaintype lattice
                  -> domaintype lattice;
   val narrow : domaintype lattice * domaintype lattice
                  -> domaintype lattice;
   val print  : domaintype lattice -> unit
end;
```

The definition for `alpha` should be read as "`alpha` is a function taking a `bool lattice`[1] argument and returning a `domaintype lattice` value". The definition for `equal` should be read as "`equal` is a function taking a 2-tuple of `domaintype lattice` values and returning a `bool lattice` value".

The `alpha` and `gamma` functions directly correspond to the abstraction and concretization functions defined in Section 3.3.1 while the `pwided` and `rwided` functions correspond to the precise and relaxed widening operators. The purpose of the `narrow` operation will be discussed when we discuss "splitting" scopes in Section 6.2. The `alpha` function takes a value in the concrete domain for `boolean` and produces something that is a `domaintype lattice`. The signature guarantees that `domaintype` is in fact a type, but that is the *only* restriction in the signature. This permits the definition of arbitrary abstract representations (subject to the semantic constraints that cannot be checked by the type system).

[1] By `bool lattice` we mean the lattice representing the concrete domain for boolean values, we do not mean a "boolean lattice" as defined in lattice theory.

The signature "cheats" a bit in the definition of the `equal` function; `equal` is required to return a `bool lattice` value rather than an abstract `domaintype lattice` value. This choice in the interface was made purely for convenience in the interpreter — there would be no semantic difference in having a `domaintype lattice` value returned and having the interpreter apply `gamma` after `equal` every time that `equal` was applied. The signature also requires that there be an output routine for the abstract values (primarily for debugging purposes).

Notice that the signature does not require abstract definitions for the normal boolean operations such as `and`, `or`, `not`, etc. These operations do not need direct support in the abstract domain as they are not primitive in the interpreter; they are defined by Scheme code that only relies on conditional expressions and `equal` over boolean values.

The signature for abstract integer values is somewhat more complex due to the number of primitive integer operations defined in the system. We will omit the type signatures of each operation for clarity.

```
signature INTBASETYPE = sig
   type domaintype;
   val alpha : ...
   val gamma : ...

   val distinct : ...
   val equiv  : ...
   val equal  : ...
   val pwided : ...
   val rwided : ...
   val narrow : ...

   val muld : ...
   val subd : ...
   val addd : ...
   val negd : ...
   val led  : ...
   val leqd : ...

   val split_range : ...
   val print : ...
end;
```

In this signature, we require functions for the primitive operations, and introduce `distinct` and `equiv` functions in addition to the `equal` function. The `equal` function determines whether two domain values represent the same set of potential values. The `equiv` function determines whether two domain values are *guaranteed* to represent the same value at run-time. Using the integer interval sets, this guarantee can only be made when both domain values consist of the same single integer. Consider the overlapping intervals, `[4..5]` and `[5..6]` — we cannot *guarantee* that the actual run-time value will be 5 in each case. The `distinct` function determines whether two values are guaranteed to represent different run-time values. This condition can be satisfied for integer interval sets if we know that there is no overlap in the two sets of intervals. For example, using the integer intervals, the interval `[4..4]` and `[5..6]` are distinct, but `[4..5]` and `[5..6]` are not. The `pwided` and `rwided` operations are the widening operations for the domain; the `narrow` operation will be discussed when we discuss "splitting" scopes in Section 6.2.

The next set of functions represent the abstract versions of each primitive operator. Each function takes as arguments and returns abstract domain values. In this case, we require support for multiplication, subtraction, addition, negation, less-than comparison and less-than-or-equal comparison. Note that we do not support division as a primitive operation since we are not supported floating point numbers in the prototype.

The `split_range` function takes a conditional expression operator and two values and returns an abstract value that represents the "part" of the first value that satisfies the conditional. So, using integer interval sets, giving `split_range` the conditional "less-than" and the interval sets {`[1..5], [10..15]`} and {`[6..9]`} would produce the result {`[1..5]`}. See Section 6.2 for details on the issues involved with splitting values.

6.1.3 Changing Abstract Domains

Due to the structure of the system, we have found that changing abstract domains is a fairly straightforward exercise. For example, we were able to build a "normal" Scheme interpreter in our system simply by creating a set of abstract domains that were identity mappings. The only difference between the semantics of the resulting interpreter and a "normal" semantics is that undefined values are treated as $\top$ rather

than causing errors.

6.2 *Split*ting Scopes

Recall that in Section 3.5.3 we assumed the existence of a *splitting* function. The purpose of a splitting function for a particular domain is to take advantage of value constraints that can be inferred from conditional expressions. For example, given an expression such as `(if (> x 5) (f x) (g x))` we would like to take advantage of the information that the value of `x` during the evaluation of `f` must be greater than 5 and that the value of `x` must be less than or equal to 5 during the evaluation of `g`.

There are two main issues involved in building a correct set of bindings given a conditional and an old set of bindings. First, we must be able to split an abstract value into a "true" and "false" partition given some simple comparison involving the value. Second, we must be able to merge sets of bindings that are generated by comparison operations joined by boolean connectives.

Value partitioning is performed by a combination of abstract domain operations and interpreter transformations. Each abstract domain interface contains a `split_range` function. This function takes a comparison operation identification and a pair of abstract values and returns an abstract value consisting of *at least* the partition of the first abstract value that makes the condition true. For example, a (stylized) call to `split_range` for integer intervals might be `(split_range '<' {[1..5]} 3)`. Hopefully the result of this call would be {[1..2]}; this is the case in the implementation. Note however, that another valid partition would be {[1..5]} since that range contains *at least* the values {[1..2]}. The value {1} would not be a valid partition as it does not contain the value {2}.

The definition of `split_range` does not require the abstract domain to know anything about arbitrary combinations of conditions, the only knowledge required of the abstract domain is knowledge about comparison operations on the abstract values. This isn't really an additional "knowledge" requirement on the abstract domain, but is simply an requirement that the interpreter must be able to extract more information than a boolean regarding how the abstract domain evaluates conditions on abstract values. Note that it is always safe for `split_range` to be implemented as an identity operation; such an implementation simply sacrifices overall accuracy.

Dealing with the composition of conditions is the role of the interpreter. When the interpreter encounters a condition such as `(< x y)`, it uses the simple `split_range` function to construct the appropriate set of bindings. For the `(< x y)` condition, the interpreter needs to build a "true" scope containing two bindings. One of the "true" bindings relates x to the result of `(split_range "<" (find x) (find y))` where `(find x)` returns the abstract value for x in the current set of bindings. The second "true" binding needs to calculate the binding for `y`. The interpreter knows about the semantics of comparison operations and so can perform a simple transformation on the condition in order to make use of `split_range` to calculate the binding. The correct binding for `y` is calculated by `(split_range ">" (find y) (find x))`. The "false" bindings are calculated in a similar way; the interpreter inverts the conditional expression and calculates the bindings. The interpreter optimizes the binding calculations such that only comparisons that create binding information are generated. For example, in the comparison `(< 5 x)` we only calculate the "true" binding for x using `(split_range ">" (find x) 5)` since literal values never have bindings. It is important to observe the "division of labour" here — the abstract domain is only responsible for the semantics of comparisons on abstract values, the interpreter is responsible for the semantics of relationships *between* operators.

The final aspect of performing a general scope split is to coalesce the bindings generated from a composition of comparisons. For example, in order to correctly calculate bindings for the condition `(or (< x y) (< x 10))` we need to merge the bindings from each of the conditions. In Scheme, we need to worry about the `and` and `or` operations (`not` operations are handled by expression transformation). If we have an `or` in an expression, the overall binding for an identifier is simply any value represented in either binding for the identifier generated by the two subexpressions. This calculation is exactly the behaviour of the precise widening operation. Thus whenever the interpreter encounters an `or` during a split, it simply evaluates each subexpression and then generates bindings for each identifier by precisely widening the bindings for the identifier generated by any subexpression. It may be the case that a particular identifier only occurs in one clause; in such cases no widening occurs (or alternatively, the binding is widening by $\perp$).

Splitting a conjunctive expression involves binding an identifier to the set of values represented in all of the subexpressions. This type of abstract value merging has not been used elsewhere; the `narrow` operation is present in each interface to

provide this functionality. The requirement for a `narrow` operation is that the value generated by narrowing two abstract values contains *at least* the values that are represented in both original abstract values. Thus, as with other operations, it is safe to choose an identity operation for `narrow`, although we would normally expect `narrow` to be equivalent to intersection for set valued abstract domains.

We know of no other partial evaluation algorithm that attempts to refine abstract value bindings across branches of conditional statements. Normally, the reason for this is that fixed-height lattices are used to represent primitive types and that no meaningful information could be represented by a such a splitting operation.

6.3 Improving Residuals

There are a number of issues related to producing "good" residuals that are not addressed in the formal algorithm presented in Chapter 3. Although these factors do not fundamentally effect the correctness of the residuals, they do have a direct impact on the applied usefulness of the techniques. Many of these factors are related; one needs to evaluate aspects of all of them in order to produce high-quality residual programs. The prototype implementation makes simplistic choices in most cases; the particular choices made will be discussed for each topic.

There are two concepts that are referenced several times in the following discussion: the idea of a function *closure* and the idea of a *continuation*. The basic idea of a function closure is that a closure encapsulates all of the dependencies that a function has with its environment. For example, if a particular variable is free within a particular function but is defined by an enclosing function, that variable is part of the closure of the inner function.

The idea of a *continuation* is a bit more unusual. A continuation is a function that captures the "rest of the computation". For example, consider the following simple expression:

```
(lambda (x) (+ 4 x))
```

The continuations form of the expression is as follows:

```
(lambda (k x) (k (+ 4 x)))
```

In this case, the identifier `k` is the continuation for the function; when the function is applied, the remainder of the computation is captured by `k`.

Continuations make control flow dependencies explicit — if the result of a computation is used in a subsequent computation, the subsequent computation will exist as part of the continuation for some evaluation of the first computation.

6.3.1 Memoization

Recall that memoization was introduced in Section 2.5. The basic idea of memoization is to create sets of equivalence classes for functions where each equivalence class maps between some function closure and a particular residual. Given a particular function f with a closure c, before partially evaluating f, the algorithm must determine if there is an existing residual f' with closure c' that can be used. There are two basic issues determining whether f' *can* and *should be* chosen. The first issue is the relationship between the information in the closures c and c'. The second issue is how much of the information in the closure of c' was used in determining the residual, or in other words, how much useless information there is in c'.

The normal requirement for choosing to reuse a particular memoized function, f', for a possible specialization of f, is that the closure of f' must be identical to the closure for f. This rule is not the only safe choice; our algorithm guarantees that given *any* closure (environment) below the memoized closure, the result of the memoized function will be safe to use. This means that we could define the specialization rule such that we only specialize if there is no current specialization with a closure above the current closure. Requiring closure equality means that any difference in the closures of f and f' disqualifies f' from consideration, even if the difference in the closures does not have any bearing on the result of the specialization. For example, consider a function like *sin*. The result of $sin(x)$ is between 1 and -1 independent of the value of x. Using closure equality, if *sin* is specialized with an x value that is bounded to the range 0 to 360 (degrees) and is then specialized again with a value in the range 360 to 720, the first specialization would not be reused even though there is no difference in the range of potential values. Alternatively, choosing to not specialize when the closure for f is below the closure for some f' can also be a problem. For example, if we encounter a call to *sin* with the parameter having the range 0 to 90, we would not create a specialization if there was an existing specialization of *sin* for the range 0 to 360. In general, using the "below" rule, we would lose many specialization opportunities if we encounter partially static function calls before fully static function calls. Making this entire issue even more difficult is that it is not always

optimal to unfold or inline computations at every opportunity; expanding the code through inlining by some additional factor does not guarantee faster code and may in fact increase running times due to cache effects, memory utilization, and other factors.

The closure "equality" choice and the closure "below" choice form the boundaries of an entire range of rules. For example, we could define some distance metric and choose to reuse an existing specialization if it was within a particular distance of the current closure. Another alternative would be to *always* specialize if the new closure contains values that have a different value when concretized into the concrete domain (recall that $\top$ represents all values in the concrete domain that are not directly representable in the natural concrete domain). If a value in the new closure has a different concrete domain value then there is likely to be some "real" difference that we can take advantage of during the specialization phase. Finally, we could include the set of values that were actually used during the specialization phase as part of the memoization. The choice about whether to select a possible residual could then be restricted to those bindings that actually influenced the specialization of the memoized function.

Any potential solution for this problem runs the risk of either over-specialization or over-generalization. Off-line partial evaluation generally resolves the problem by allowing users to intervene and directly change the BTA annotations. Unfortunately, this approach becomes an "all-or-nothing" choice; either all of the effected code will be inlined or none or it will since BTA annotations don't reflect the idea of limited inlining. Andersen [7] briefly discusses the idea of *k-limited* annotations that allow the specializer to restrict recursive inlinings to k levels. Andersen restricts k to 1 in his thesis and has not investigated ways of automating the choice of k. Even with this approach however, the choice of k is fixed on a global basis; it seems clear that effective partial evaluation should dynamically vary the amount of inlining during specialization.

Ruf defines a *domain of specialization* [70] or *DOS* for a particular residual to be the set of values for which the residual and the original function have the same behaviour. It is important to differentiate this statement from a soundness statement; this is a broader statement than the requirement that the residual and the original program have the same behaviour on the abstract values used to create the residual. In general, the domain of specialization for a residual will be a superset of the

values used for specialization. Ruf then characterizes *optimal re-use* as choosing to re-use a function if the concrete values represented by a particular abstract value for the argument of the new call are a subset of the DOS of the memoized function *and* if it is not the case that the DOS of the new residual is a subset of the DOS of the existing residual. Using our definition of *below*, this means that the argument value is below the corresponding value of the existing residual and that the DOS of the new residual is not below the old residual. The intuition is that the DOS characterizes the properties (or values) of the abstract argument value that are actually "used" during the specialization. Casting this as a behavioural statement, Ruf's optimal re-use statement requires that the new residual has the same behaviour over its set of arguments as the memoized residual. This is somewhat similar in flavour to a contravariant typing statement.

As Ruf observes, an exact DOS is undecidable but can be approximated. Ruf introduces an additional calculation to estimate the DOS as part of his strategy. The basic idea of calculating the DOS is to define a second evaluation that is performed in parallel with the normal evaluation. At each step the DOS calculation determines the most general value that satisfies the current calculation. As calculations use more information about a particular set of values, the DOS is lowered in the lattice. Ruf's DOS calculation is eager and as such, can be overly conservative in certain instances. For example, if a parameter is involved in any `let`-bound calculation that subsequently becomes dead code, the `let`-bound calculation can change the DOS.

Implementing an equivalent DOS calculation in our system would be reasonably straightforward. Essentially, we would only need to define DOS values for primitive operations within each abstract domain and propagate DOS bindings through the interpreter. As this would be a minor change to our domain requirements, implementing the Ruf's algorithm for selecting residuals would be straightforward.

The DOS approach has much of the same range of choice as the "closure below" choice discussed earlier. The accuracy of the DOS approach is directly related to the accuracy of the estimate for the abstract values actually used in the specialization. The modularity of our domain requirements makes this factor a reasonable parameter in the design of the abstract domain; if fewer residuals are desired, simply return more specific values than necessary as the DOS estimates from the primitives.

The prototype system adopts the simple closure equality strategy. This implies that we can in fact generate duplicate functions within a program residual. The

immediate plans are to change the algorithm in two ways. The first change will be to compare a resulting residual to the memoized residuals when a specialization is actually performed. If the residuals are equivalent then we will not introduce the new memoization entry but rather reuse the existing memoization. This means that we would waste the time spent evaluating the function, but would decrease resulting code size. The second goal is to implement a domain of specialization technique for memoization choices.

6.3.2 Code Duplication

The possibility of creating duplicate function instances is not the only problem related to code duplication. In general, inlining residual computations can cause computations to be duplicated. Consider the following example (modified from Jones [46]):

```
(define f
    (lambda (n)
        (if (= n 0)
            1
            (let ((y (f (- n 1))))
                (+ y y)
)   )   )   )
```

The `let` binding captures the value of the recursive call and returns the doubled value. If this function were blindly unfolded for an unknown `n`, the following function would result:

```
(define f
    (lambda (n)
        (if (= n 0)
            1
            (+ (f (- n 1)) (f (- n 1)))
)   )   )
```

In this case, replacing the identifier `y` with its residual computation is a poor choice; the resulting algorithm requires exponential time compared to the original linear time algorithm.

The basic rule for both an on-line and off-line evaluator is the same: do not permit a function call to be duplicated in the same branch of code. Off-line specializers typically adopt a two-stage specialization strategy (Jones [46]) to avoid this problem. In

a two-stage specialization process, a static annotation means that a function *may be* unfolded rather than that it definitely *will be* specialized. The determination about whether to actually unfold is made not at BTA time, but at specialization time. Bindings for identifiers are called "duplicable" by Jones if there exists a path through the related expression in which the identifier occurs more than once. The specialization decision is then made by checking whether the duplicable identifiers are constants or identifiers. All non-trivial bindings effectively transform annotations to *dynamic* for that particular specialization.

The on-line decision is similar in the sense that the determination about duplicable identifiers must occur and that a particular in-lining decision depends on the residual for duplicable identifiers. The main difference is that in the on-line approach this fits naturally with the overall evaluation algorithm. The on-line specializer already considers changes to annotations, while in the off-line approach this type of decision is a fundamental shift in approach and forces the off-line algorithms into adopting a partially on-line approach.

The prototype system implements conditional inlining and unfolding based on whether identifiers are duplicable. The analysis consists of performing a count along each path through the expression (an operation that is strictly local to the body of the function or `let`-expression) and determining if each identifier occurs more than once. This count is performed in the same way for both `let`-bound identifiers and formal function parameters. When the binding for each identifier takes place (i.e. at the beginning of a function evaluation and at the beginning of a `let` statement) the identifier is added to a list of unsafe identifiers if it is both duplicable and a non-trivial computation. In the prototype implementation, non-trivial computations are a bit more general than simply constants or identifiers — the prototype allows the duplication of any expression that does not involve a non-primitive function application or a potential side-effect (see Section 6.3.3). Finally, when an identifier is encountered, if it is not on the list of unsafe identifiers, it is replaced with its residual, otherwise it is not replaced.

6.3.3 Computations with Side-effects

Impure computations are a substantial problem in any partial evaluation algorithm. In general, it may be difficult to determine whether a particular identifier has an

alias; in fact, in languages such as C in which aliases can be created at will, exact alias analysis is undecidable. There are two basic approaches that can be taken: first, one can restrict the model so that all aliased store either has known alias relationships or is treated as dynamic. This approach is adopted by Nirkhe and Pugh in their partial evaluator for hard real-time systems (discussed in Section 2.6.3). The second main option is to track sets of alias relationships. Sets of alias relationships provide essentially the same information as a "may-alias" analysis [53] [29]. Andersen [7] has implemented a simple form of pointer alias analysis in his partial evaluator for C. His analysis does not track conditional alias relationships but basically finds the union of all possible alias for each aliasing variable within functional units. Every model has problems with handling truly unknown aliasing relationships — if the set of potential aliases becomes unknown, that destroys nearly any further specialization since *every* memory location must become unknown.

An additional problem is related to the issue of code duplication in the previous section; it is generally not safe to duplicate any computation that involves a side-effect. As mentioned in the previous section, the prototype evaluator handles this issue by not duplicating any code that involves an imperative feature. While this is a reasonable choice in languages in which the use of imperative features is rare, it clearly is not acceptable in languages such as C.

The final issue relates to merging of run-time state after conditional expressions. Consider the following expression (assume `c` is dynamic):

```
(if c
    (begin
        (set! x 5) x )
    (begin
        (set! x 7) x )
)
```

Although we can replace the two references to `x`, we cannot remove the assignment statements. Consider the following incorrect residual:

```
(if c 5 7)
```

This expression has the same value result as the original expression for all input. It is not a correct residual however, since the state of the system after this expression is not going to match the state of the original expression for all input. The standard approach is to insert "explicators" [61] in the residual code. Explicators are simply

assignments that guarantee that the state of each branch matches the abstract state at the end of the computation. In off-line systems this is a larger issue since this implies creating a run-time assignment based on compile-time values. This resembles the partially on-line decisions made for potential code duplications as discussed in Section 6.3.2. Note that in general there may be a large number of assignment statements replaced by a single explicator. For example, if there was a `(set! x 9)` following the use of `x` in the first branch of the example, the `(set! x 9)` would remain as the explicator, but the `(set! x 5)` could be removed entirely.

In off-line systems explicators only need to be added at the end of dynamic conditionals when the value abstractions are not guaranteed to generate the same run-time value. In the proposed on-line system there is an additional case: if an assignment uses a value for which the concrete domain representation is not a definite value, then the run-time behaviour is unknown and the assignment must remain in the residual. All other state changes that involve definite values can be removed since they are unconditional along that evaluation path. Note that comparable assignment statements in an off-line system would also remain since such statements would necessarily be annotated as dynamic due to the fact that the compile-time state of the variable does not have a single value. The on-line system has an integrated decision process rather than the partially on-line approach used by the off-line approaches.

Imperative features are not the main focus of our work and as such, the prototype implementation avoids most of these issues by always leaving imperative code in residuals. In other words, even if all imperative statements could safely run at compile-time, our current system will leave them in the residual. The system will correctly use values that are created by imperative features, but will not generate minimal residuals in code with imperative features. Future versions will adopt the abstract store model and set-based alias analysis; such a model is consistent with the overall approach adopted in the abstract domains.

6.4 Other Language Issues

6.4.1 Arity Raising

The general term *arity raising* refers to transformations that increase the number of parameters to functions. In off-line partial evaluation, *arity raising* refers to the

process of separating the static and dynamic portions of a partially static structured type into several parameters. Arity raising has similarities to Launchbury's projections based approach that was discussed in Section 5.2.2. The static projection of a function that takes a partially static structural parameter is parameterized by the static portion of the structure while the dynamic projection is parameterized by the dynamic portion of the structure. Effectively, this is raising the overall arity of the function even though the arity of each projection may remain the same.

Arity raising in the traditional off-line sense is not directly applicable in an on-line approach as no actual structural decomposition is necessary. There is however, a different view of arity raising that can be useful in the on-line approach. It is reasonable to view the closure of a function as an implicit parameter, or in fact, as a series of implicit parameters. If all functions were "flattened" such that every closure variable was passed in an explicit parameter, this would permit finer granularity decisions regarding the effective annotation of the closure. In [60], Mason defines a continuations based intermediate language that performs such a flattening as one stage of the compilation process.

There are two major benefits for performing arity raising by flattening within an on-line partial evaluator. First, as already mentioned, such a transformation would allow the evaluator to make finer-grained decisions regarding the equivalence of function memoizations. Second, flattening would more easily allow for the identification of relationships between bindings within function closures since these relationships would be explicitly present within the call graph.

The prototype implementation does not perform any form of flattening. As noted, structural flattening provides no benefit to the on-line algorithm with the structural domain model presented in Section 5.2. Although closure flattening may provide some benefits, this remains as future work, possible by using Mason's flattened intermediate form [60] as the basis.

6.4.2 Complexity of Semantics

The conceptual complexity of building a partial evaluation framework for a given language is strongly related to the conceptual complexity and semantic definition of the source language. Writing a partial evaluator is *more* complex than writing a

normal interpreter; as mentioned in Section 6.1.3 a standard semantics interpreter is a special case of the partial evaluation framework that we have defined.

Consider applying our partial evaluation technique to languages such as C. The memory model in C is closely related to real machine memory layout; there are requirements on the behaviour of pointer comparisons, the layout of structures, etc. The semantics of these operations would have to modeled within the interpreter in order to correctly calculate the value of expressions. Unfortunately the semantics of some operations within C are not completely defined. For example, ANSI C [1] requires that type `long` be able to represent at least the integer values representable by the type `int`. A compiler conforms to the standard if it chooses to define the two types as structurally identical; a C-to-C partial evaluator could only choose to transform based on the requirements in the standard. If a C-to-C partial evaluator produced C code based on the assumption that `long` and `int` were structurally equivalent, the produced C code no longer be ANSI C conformant if the assumption was exercised within the residual.

As discussed in Section 2.6.2, Andersen's approach to these issues is to produce generating extensions [5] [6] and then to have these generating extensions produce the actual code that would then be compiled. Meyer [61] performs a deeper interpretation but does so in a restricted language with a much simpler semantics.

Although both off-line and on-line algorithms must implement a safe approximation to language semantics, the requirement for off-line evaluators is somewhat weaker than for on-line algorithms. Off-line BTA is a fixed-point calculation that depends only on a simple abstract semantics of the source language involving *static* and *dynamic* annotations. Although in reality, accurate off-line BTA relies on a reasonable model for alias relationships, such an approach can avoid implementing a safe abstract semantics for the entire language. On-line evaluators must implement safe semantics for the entire language. Although neither off-line nor on-line approaches need to be "complete" in the sense that any aspect can be treated as unknown and will cause a safe approximation, the overall effect of choosing unknown can be to greatly reduce opportunities for specialization.

6.4.3 Separate Compilation

Partial evaluators generally assume that the entire program is available at partial evaluation time. This assumption relates to both binding-time analysis issues and specialization. Although for convenience the discussion will deal with the issues separately (adopting an off-line bias), the concerns apply equally to on-line systems.

First consider binding-time analysis. The basic goal of binding-time analysis is to safely annotate program variables and function calls as being either *static* or *dynamic*. In terms of variables, *static* means that all possible values for the variable are available at compile-time. If a language does not support any form of side-effects, the binding time analysis can easily be performed separately as long as the BTA assumes that any value returning from an separate module is dynamic. A more accurate estimate can be made by adopting Andersen's approach and introducing *binding-time signatures* [7] for each module. Such signatures provide other modules with information about what annotations have been made. Unfortunately, in the presence of mutually recursive modules, such an approximation is going to be extremely conservative unless the mutually recursive modules are analyzed at the same time or provide explicit symbolic information regarding the external dependencies if a dynamic annotation is made solely due to an external module. Andersen's approach follows the independent analysis route, assuming that all external module values are dynamic. Andersen did not propose any form of symbolic dependency analysis. No one else has attempted to address languages with explicit modules and completely separate binding-time analysis.

In addition to the relatively simple problem of dealing with a pure language, in general a system may need to handle languages that permit cross module side-effects. As one extreme, consider C "modules". A C program is permitted to cause side-effects in *any* externally visible variable. In addition, if any of these externally visible variables are pointer types, then without a fairly accurate value analysis, a BTA would be forced to assume that *any* variable that had its address captured could become dynamic after any function call outside the current compilation unit. Ever achieving reasonable binding-time results would be unlikely in such a model. Languages such as Ada [80] are somewhat easier to deal with since the package interfaces contain more definitive information and reference coercions are much more tightly constrained than in C. Even with the better interface however, a BTA would be forced to treat as dynamic any `input-output` parameters or exported non-constant variables.

The second aspect within partial evaluation, namely specialization, is even more problematic in such environments. By definition, a specialization must be able to determine the actual values for variables that are static within the evaluation. In order to determine static values that result from interactions with external modules, the specializer will have to be able to perform evaluations of the external module. It does not seem possible to resolve this issue without either abandoning separate compilation or being satisfied with purely local specializations. It is likely that to be highly effective, the application of partial evaluation techniques in general environments would need to occur as post-link optimizations. Early linking as suggested by Mason [60] may help to alleviate such problems.

6.4.4 Exceptions

Exceptions permit run-time branching decisions to be made. From the perspective of a partial evaluator, exceptions have similar characteristics to both continuations and first-class functions. Raising an exception requires that the current evaluation be terminated in favour of an expression that performs some recovery action. Raising an exception is similar to calling an alternate continuation that includes the recovery evaluation before continuing the computation at the appropriate point. Determining the effective binding-time annotations when exceptions can occur involves considering all possible control flow paths from a given point raising the exception to any point that might catch the exception. This is similar to determining the set of call sites for a first-class function and generally requires some amount of value analysis in order to preserve accuracy.

Fortunately, most exception handlers are "well-behaved" in the sense that they either restore the state of (part of) the computation, provide a default value so that computation can presume, or terminate the computation completely. In any of those cases, the exception is unlikely to change the state of annotations, so conservative approximations should be reasonable. Although various people have investigate higher-order functions, there has been no direct work on supporting exceptions.

A related language feature is *call-with-current-continuation*. Call-with-current-continuation, or *callcc*, allows a programmer to explicitly capture the continuation of a particular program point and to pass that continuation as a parameter to a subsequent function. Calling that continuation is similar to raising an exception: the

execution continues at the point immediately following the site of the original *callcc*.

6.4.5 Compile-time features

Purely compile-time language features generally do not impact the partial evaluation process. The basic reason is that partial evaluation transforms run-time operations into compile-time operations; if an operation has only compile-time semantics, there is little effect on the partial evaluator. Examples of such features are static type checking, generics, and ad hoc polymorphism (overloading).

Partial evaluators generally assume that any static typing issues have been resolved prior to partial evaluation (i.e. that the program is valid). Static types then only concern a partial evaluator as a mechanism for providing the evaluator with additional constraints or annotations that can be applied during the evaluation process. For example, the partial evaluator can take advantage of the fact that identifiers that define constant values will never change annotations after being created since the language guarantees that the values cannot be modified after creation. In fact, in languages that permit only compile-time values for constants, constants can always be treated as static.

A *generic* [80] or *template* [79] routine is a code fragment that is parameterized by type and/or value information. Conceptually, instantiating a generic with particular information creates an instance of the related code suitable for use with the given type and/or values. The overall effect of generic instantiation can be similar to partial evaluation — incorporating specific information into more general code and producing an optimized version that takes advantage of the particular static information. In reality, implementations either simply "box" parameters and use a single un-specialized version of the code or simply duplicate the code in the same fashion as a macro expansion. Partial evaluation fits naturally into this framework by accepting the instantiated generics (either the sharing code or expanded form) and applying the normal evaluation process.

Compile-time overloading is similar to generics as far as partial evaluation is concerned; partial evaluation assumes that the resolution has already succeeded and that all that remains is the normal partial evaluation process. The only exception to this assumption is when languages require run-time overloading resolution; such

models are effectively the same as limited object oriented models as was discussed in Section 2.6.1.

6.4.6 Applying Heuristics

Heuristics have long had a major role in real compilers. These assumptions are realized in many ways: peep-hole optimizations, loop unrolling, branch prediction, size/space trade-offs, register allocation, instruction scheduling, and so on. Each of these optimization techniques have a solid basis in real performance issues and have generally been studied extensively. As a simple example, new architectures may be tested in an exhaustive manner to find code sequences that have the same effect of other code sequences (even if the designers of the architecture did not foresee such an equivalence) in order to improve peep-hole optimization.

Partial evaluators have traditionally ignored these issues and focussed on transformations that are set in a more formal semantic framework. There are however, potential opportunities that should be explored. For example, one interesting method for combining register allocation and instruction scheduling is *code coagulation* [49]. The basic idea is that the code "hot-spots" should be compiled independently and be independently given free choice of registers. At each point in the code production, the next "hot-spot" is chosen; when these locations meet, any necessary register transfers are introduced. Adopting this approach can reduce the number of register spills required within the most frequently executed code, thus improving overall performance.

On-line partial evaluation may be a reasonable method for estimating code "hot-spots" and performing code coagulation. On-line partial evaluators follow the interpretive flow of control in the source program and produce residual code after evaluating each expression. Residuals for code that is deepest in the evaluation is normally encountered earlier in the evaluation process so performing register allocation during partial evaluation may be a reasonable approximation of the code coagulation approach. Alternatively, partial evaluation could provide "expected profile" information for use during code coagulation. One area for further study would be to investigate the predictive power of such estimates when compared with real execution profiles.

Chapter 7

Conclusions and Future Work

7.1 What's New?

This thesis has presented a new algorithm for on-line partial evaluation. The separation and characterization of general abstract domains is an important improvement in that it allows domain design to focus on the abstract information that is desired rather than on the evaluation algorithm. The algorithm itself has been proven to terminate and to generate sound solutions based solely on the general characteristics of the abstract domains used by the algorithm. A key improvement in our approach is using both precise and relaxed operations when manipulating abstract values. Comparable systems compute all collections of abstract values using a single approach — least-upper bounds in finite lattices. Our approach preserves termination while allowing substantial improvements in the accuracy of the analysis. We have carefully defined the termination and soundness characteristics for the on-line partial evaluator with respect to the standard semantics. Characterizing termination and soundness as relationships between the environment during partial evaluation and environments under the standard semantics provides an intuitive basis for reasoning about the residuals. We believe that such characterizations will be critical to the application of partial evaluation techniques in a wider context.

The abstract domains presented in Chapter 5 capture more static information than the simpler lattices used in other approaches. At the same time, these domains retain reasonable convergence properties. We have shown that these particular abstract domains satisfy the formal requirements for domains. This modular approach

to proving algorithm properties is useful for reducing the work involved in developing the proofs without sacrificing confidence in the results. Although it has not been discussed throughout the thesis, the modular design also allows the domains to be designed with little concern for the actual language being interpreted. Although some language specific domains might be necessary, common abstract domains such as integers, booleans, etc., should be reusable between implementations of the our algorithm for different source languages.

We have built a basic proof of concept prototype. Though the prototype made simplistic choices regarding many of the issues related to residual production, it has demonstrated the viability of the analysis phases and the ease of changing abstract domains.

7.2 What's Next?

This thesis has focussed primarily on the theoretical aspects of the proposed framework. One of the main directions for further work is to work towards a framework for the "applied" aspects of on-line partial evaluation. In particular, characterizations need to be developed for profitable code expansions through the use of either estimates of code behaviour or limited profiling information. It would be particularly interesting to investigate the amount of profiling needed to "inform" the abstract analysis about sufficient program characteristics to allow for good specialization decisions.

There are three major areas of future work. The first area is in improvements to the algorithm and the models used for abstract domains. The second area of future work is in applying other abstract models and characterizing the types of information needed to apply these models. The third major area is in direct applications of these techniques to solving traditional compiler problems and in discovering other types of useful information. In particular, using these approaches to characterize typical programs could be useful in determining profitable avenues for further optimization strategies. The following sections briefly summarize future directions.

7.2.1 Foundations

Improvements to the On-line Algorithm

There are three changes to the presented algorithm that need further study. The first two changes are with respect to the d parameter in the algorithm. Recall that the intuitive meaning of the d parameter is that it reflects computations that are potentially divergent. Currently, the algorithm is very conservative with respect to potentially divergent computations; as soon as a dynamic conditional is encountered, *all* derived computations are assumed to be potentially divergent. This assumption is often overly conservative. In particular, when we discover additional results that are not in $\xi(\lambda x.\, e_{id})$, we currently re-evaluate the function application with the expanded ξ. In order to improve accuracy, we can actually perform the re-evaluation with "d = false" rather than "d = true". Essentially this change is adding a hypothesis that the expanded ξ will not lead to divergence. If, with the expanded ξ, a conditional turns out to be dynamic, we simply regress to the conservative case and could further expand ξ and δ.

A related change to the overall algorithm that we intend to investigate involves changing d from a simple boolean into a vector of booleans with one flag for each function. Consider a computation such as the following:

```
(if x (fact 5) 0)
```

where x is unknown. Under the current strategy, the evaluation of `(fact 5)` occurs with d = true. If the integer domain converges rapidly, as does our interval domain, the result of this computation is the interval $[0..\infty]$ rather than the accurate result $\{0, 120\}$. The reason for the rather poor estimate is that we assume that the subexpression `(fact 5)` could be divergent. However, in reality, such functions are only divergent if there is a dynamic conditional in the derivation between the outermost evaluation of `fact` and a derived evaluation of `fact`.

The third change that we plan to investigate is related to how we produce values when we find fixed-points. Currently, when we realize that our latest result does not extend ξ, we produce that result immediately. Again, this is a fairly conservative approximation technique. The basic intuition is that we expand our estimate until we pass the ideal result; however, in the current algorithm we then simply return the computed overestimate. It may be possible to refine our estimate and, in effect,

reduce the size of our estimate and try to get closer to the ideal result. One possible approach for reducing the "size" of the estimate is to increase the accuracy of the formal parameters to functions. The proposed change is to produce the result of

$$\mathcal{P}[\![e]\!]\,\rho[x \mapsto < e_1^{\alpha}, e_1^{R} >]\,\delta'\,\xi\,\text{true}$$

as the result in line 3.15(11) when our latest result estimate does not expand ξ. Currently the algorithm simply produces e^{α}. The e^{α} estimate is likely to be a substantial overestimate of the actual result however, since the value bound to x during the production of e^{α} includes *all* of the values in δ'. However, since the new result estimate is subsumed by ξ and at this point we know that the value of e_1^{α} has been calculated with respect to ξ, we should be able to improve our result estimate by using only the e_1^{α} portion of δ' to find the overall result.

All three of the above changes were motivated by properties of the proofs presented for the current algorithm. As such, we are fairly confident that all of these changes would preserve the correctness and termination of the algorithm. We do not, however, know which of the changes would be profitable in terms of the tradeoff between increasing the time for the analysis versus the expected increases in accuracy. In order to fully explore these aspects of such changes, we need to move beyond the current prototype implementation into a more fully developed system. As such, this remains as future work.

The other major set of changes needed in the current implementation is to bring the memoization and residual production into line with approaches that have focussed on those aspects. This may be incorporated into the current implementation or may involve replacing the analysis aspects of some other system such as FUSE with our algorithm. It is not clear which of these approaches will be most viable.

Other Abstract Domains

The representation of information within abstract domains has not received much attention lately, particularly with respect to abstract structured domains. Most current approaches use some form of structural decomposition; our approach is the only exception. We would like to continue to explore aspects of representing structural information. One possibility is to allow a mix of normal structural values and functions that are sublist generators. This could, for example, allow us to append a single value

onto the end of a list of unknown length and retain information about that value. No current system models such lists.

Another aspect to consider is modeling values based on some sort of specification language. If, for example, a particular portion of code was originally specified in a specification language such as Larch or VDM, it might be useful to consider whether those specifications could be used for reasoning about what transformations *should* be safe to perform. There are substantial reliability issues involved in following this route. Arbitrary reasoning about the code and specification is not a viable approach; it is not clear whether a code transformation system should be permitted to rely on specifications during transformation rather than solely relying on user annotations and language semantics.

7.2.2 Extending the Models

Higher Order Functions

This work has not addressed higher order functions. Higher order functions are normally addressed by a combination of conservative control flow analysis and a transformation into a continuations based language. Generally, the number of functions in a program is relatively small so an on-line approach that performs an incremental control flow analysis by collecting sets of potential functions bound to particular identifiers may be a reasonable approach.

There are simple approaches to handling higher order functions within an on-line partial evaluator. Since we have assumed that there are a finite number of functions, we can form a complete lattice from the powerset of functions and use simple set union as our widening operators. This approach is completely accurate, but could potentially be computationally very expensive since *each* possible function would have to be evaluated at every application of a higher order function.

Extending the model to include first class functions removes the finiteness assumption. In such cases, it is much less clear how to construct any reasonable and non-trivial abstract model. Ideally, we would like to have a model that could reason about the types of functions that are being built. This may be possible in some very restricted situations, but does not seem to be likely in general.

Embedding Other Abstract Models

The abstract domains that we have proposed are biased towards reasoning about "values" rather than "relationships". One interesting avenue for future exploration is the possibility of using other types of formal analysis within the abstract domains. An example of such a formal analysis is constraint analysis. Adopting constraint analysis techniques as instances of abstract domain models seems to pose some difficulty. In the current approach, the interface to the domains is relatively simple and is based solely on values; an abstract domain needs to know very little about the language being interpreted. In order to embed constraint analysis, relationships within the source would have to be transmitted to the domains. Doing this in a language independent manner seems somewhat problematic, although some of the recent work in reasoning about arbitrary inductive structures is promising.

The integer list domain that we introduced incorporates a concept of "direction". Relaxed widening operations for integers are not associative since the widening operation captures the direction of growth in the domain. It is certainly possible to explicitly extend the number of directions to include, for example, the even or odd numbers as a direction, the Fibonacci numbers as a direction, and so on. It would be interesting to evaluate how many relationships in real programs could be expressed from a small set of basic relationships. Such a study could determine whether fully general reasoning was necessary for most optimization situations.

7.2.3 Applied Problems

Solving "Traditional" Problems

Some traditional compilation issues such as register allocation, instruction scheduling, control and data flow analysis, and low-level optimizations have been briefly mentioned in this thesis. One of the long term goals of developing the framework for on-line partial evaluation is to express many of these optimizations and analyses using a consistent method. Such a re-casting of techniques would be valuable to regularize the discussion of the techniques as well as expressing the techniques in a modular way. This could help in clarifying the dependencies between the techniques and reducing surprising interactions between optimization choices.

Quantifying Program Behaviour

Every partial evaluator builds models of program behaviour; these models form the basis of deciding on annotations and on how to specialize the program. Unfortunately, very little empirical data is available to use when deciding what types of models are likely to express real program behaviour. For example, in real C++ programs, how often is multiple inheritance used? What is the average size of a record in Ada? How often are methods overridden in Modula-3? Particularly in languages in which abstractions are costly, there is little on which to base particular optimization choices. There are exceptions of course; the high performance Fortran community has a fairly clear idea about the nature of such programs, but that program domain is fairly small.

Partial evaluation is not a panacea, nor can it stand alone. However, given the general movement towards higher level languages, we believe that partial evaluation can provide valuable models for program transformation.

Appendix A

Lattices

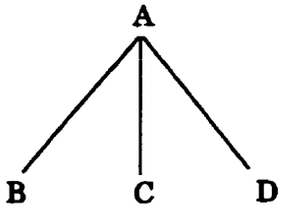
In this appendix, we briefly review some notation and the basics of lattice theory; for a complete development, we would recommend the introductory book by Davey and Priestley [30].

A *lattice* is a formal model for describing the relationships between elements in a set. A lattice is a special case of a *partial order*.

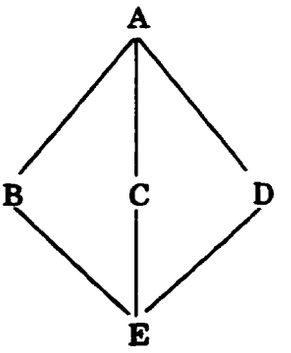
> **Defn A.1 (Partial Order)** *A* partial order $< S, \preccurlyeq >$ *is a set* S *and a relation,* $\preccurlyeq$*, on* S *such that for* $x, y, z \in S$*, the* $\preccurlyeq$ *relation is:*
>
> - **transitive**: $x \preccurlyeq y$ *and* $y \preccurlyeq z \Rightarrow x \preccurlyeq z$.
> - **antisymmetric**: $x \preccurlyeq y$ *and* $y \preccurlyeq x \Rightarrow x = y$.
> - **reflexive**: $x \preccurlyeq x$.

If $x \preccurlyeq y$ we may say that x is *below* y. Note that it may be the case that $\preccurlyeq$ does not hold at all between two arbitrary elements of S. In other words it may be the case that for some $x, y \in S$, $x \not\preccurlyeq y$ and $y \not\preccurlyeq x$. In such a case we say that x and y are *incomparable*, denoted as $x \| y$.

> **Defn A.2 (Down-set)** *Given a partial order* $< S, \preccurlyeq >$ *and an element* $s \in S$*, the* down-set *of* s*, denoted* $\downarrow s$*, is a set* D *such that for all* $s' \in S$*, if* $s' \preccurlyeq s$ *then* $s' \in D$.



(i) Lower bounds {B C D}
No Greatest lower bound



(ii) Greatest lower bound {E}

Figure A.1: Lower bounds

We can extend the meaning of a down-set by defining the down-set of a subset of S to be the union of the down-sets of the elements in the subset. More formally, given a partial order $< S, \preccurlyeq >$ and $S' \subseteq S$ then

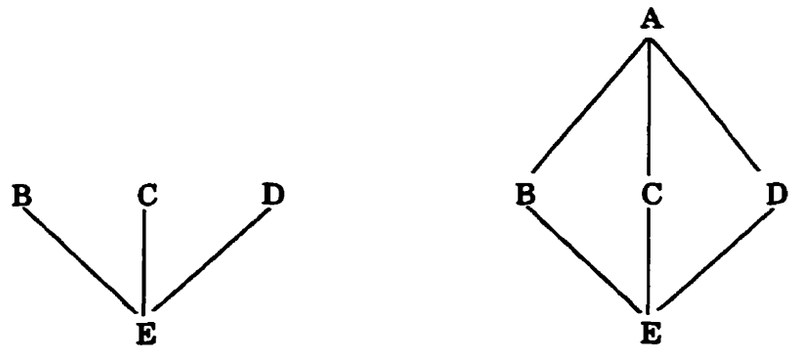
$$\downarrow S' = \bigcup_{\forall s_i \in S'} \downarrow s_i$$

It is useful to be able to talk about various *bounds* or limiting values of a subset of some partial order $< S, \preccurlyeq >$. Assume that S' is a subset of S for some partial order $< S, \preccurlyeq >$.

Defn A.3 (Lower Bound) *A* lower bound *for S' is an element $y \in S$ such that $\forall x \in S', y \preccurlyeq x$.*

Note that the lower bound of a subset of S does not have to be a member of the subset, it is only required to be a member of S.

Defn A.4 (Greatest Lower Bound) *$\sqcap S'$, the* greatest lower bound *for S' is a lower bound, y, of S' such that $\forall x \in$ { lower bounds of S'}, $x \preccurlyeq y$. We will also refer to the greatest lower bound of a set containing elements x and y as the* meet *of x and y, denoted as $x \wedge y$.*



(i) Upper bounds {B C D}
No least upper bound

(ii) Least upper bound {A}

Figure A.2: Upper Bounds

It may be the case that a set does not have a lower bound; if there are two incomparable values which constitute the set then there would be no value comparable to (and below) every element of the set.

There are symmetric definitions for "upper bounds":

Defn A.5 (Upper Bound) *An* upper bound *for S' is an element $y \in S$ such that $\forall x \in S, x \preccurlyeq y$.*

Defn A.6 (Least Upper Bound) *$\sqcup S'$, the* least upper bound *for S' is an upper bound, y, of S' such that $\forall x \in \{$ upper bounds of $S\}$, $y \preccurlyeq x$. We will also refer to the least upper bound of a set containing elements x and y as the* join *of x and y, denoted as $x \vee y$.*

As with lower bounds, an upper bound may not exist. Obviously it is the case that if no upper bound exists, no least upper bound exists.

Defn A.7 (Ascending Chain) *An* ascending chain *of elements in S is a sequence $x_1, x_2, \ldots$ such that $x_1 \preccurlyeq x_2 \preccurlyeq \ldots$.*

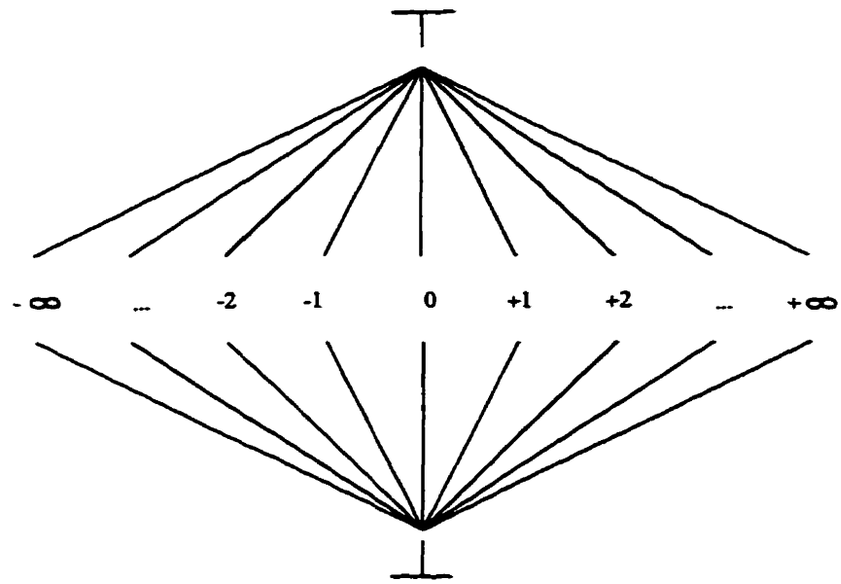


Figure A.3: Integer Lattice

An ascending chain may be infinite. We may talk about an upper bound or least upper bound for a chain, as well as for a set. As with a set, an ascending chain may not have a least upper bound (although any *finite* ascending chain will have a least upper bound).

Defn A.8 (Lattice) *A* lattice $< S, \preccurlyeq >$ *is a partial order such that* $\forall\{x, y\} \in S : x \wedge y$ *and* $x \vee y$ *exist.*

Defn A.9 (Complete Lattice) *A* complete lattice $< S, \preccurlyeq >$ *is a lattice such that* $\forall S' \subseteq S : \sqcup S'$ *and* $\sqcap S'$ *exist.*

Figure A.3 shows a lattice for singleton integer sets with the subset relation. Each "set" (a single integer value) is incomparable to any other singleton set (since no element is a subset of another), $\perp$ is considered as part of any set, and $\top$ is considered to include every singleton set.

This is a fairly simple model; the intuitive meaning of $\perp$ is "empty set" and the intuitive meaning of $\top$ is "the set of all integers".

We will often want to consider functions from $S \rightarrow S$. In particular we will be concerned with interpreting recursive functions which conceptually move through the lattice. Functions within $S \rightarrow S$ will be required to be *monotonic* and *continuous*.

Monotonic functions preserve ordering; if an element, x, is below another element, y, then the mapping of x will be below the mapping of y.

> **Defn A.10 (Monotonic Function)** *A function, f, on a lattice S is monotonic if* $\forall x, y \in S, x \preccurlyeq y \Rightarrow f(x) \preccurlyeq f(y)$.

Continuous functions preserve least upper bounds; applying a continuous function, f, to the least upper bound of a chain results in the same element as taking the least upper bound of the chain formed by applying f to each element in the original chain.

> **Defn A.11 (Continuous Function)** *A function, f, on a lattice S is continuous if, given an ascending chain* $X = x_1 \preccurlyeq x_2 \preccurlyeq \ldots$, $f(\sqcup X) = \sqcup(f(X))$.

Given a continuous, monotonic, and total function from $S \rightarrow S$, any mapping of the function is guaranteed to stay within the lattice; we don't need to worry about "falling off" the lattice. In any lattice $(S, \preccurlyeq)$, given a continuous, monotonic, and total function $f : S \rightarrow S$, f will have a *fixed point*.

> **Defn A.12 (Fixed Point)** *A* fixed point, *u, for a function f is a value such that $f(u) = u$.*

Given our lattice definition, f will also have a least fixed point[1], which will be the least upper bound of the ascending chain $\perp \preccurlyeq f(\perp) \preccurlyeq f(f(\perp)) \preccurlyeq \ldots$.

[1] As the details of the proof are not important for our purposes, we will defer to Allison [4] for the proof.

Appendix B

Concise Definitions

B.1 The Standard Semantics Interpreter

Constants

$$\mathcal{N}[\![\texttt{const}]\!]\,\varrho = \texttt{const} \tag{3.1}$$

Identifiers

$$\mathcal{N}[\![\texttt{ident}]\!]\,\varrho = \varrho(\texttt{ident}) \tag{3.2}$$

Conditions

$$\mathcal{N}[\![\texttt{(if c e}_1\ \texttt{e}_2\texttt{)}]\!]\,\varrho = \tag{3.3}$$

$$\textbf{let } c' = \mathcal{N}[\![\texttt{c}]\!]\,\varrho \tag{1}$$

$$e' = \begin{cases} \mathcal{N}[\![\texttt{e}_1]\!]\,\varrho & \text{if } c' = \texttt{true} \\ \mathcal{N}[\![\texttt{e}_2]\!]\,\varrho & \text{if } c' = \texttt{false} \end{cases} \tag{2}$$

$$\textbf{in}$$

$$e' \tag{3}$$

$$\textbf{end}$$

Primitive operators

$$\mathcal{N}[\![(\mathtt{op}\ \mathtt{e}_1\,\mathtt{e}_2 \ldots \mathtt{e}_n)]\!]\varrho = \tag{3.4}$$

let

$$\mathtt{e}'_i = \mathcal{N}[\![\mathtt{e}_i]\!]\varrho \quad \textbf{for all } 1 \leq i \leq n \tag{1}$$

in

$$\mathit{apply}(\mathtt{op},\ \mathtt{e}'_1\,\mathtt{e}'_2 \ldots \mathtt{e}'_n) \tag{2}$$

end

Function Application

$$\mathcal{N}[\![(\lambda \mathtt{x}.\mathtt{e})\ \ \mathtt{e}_1]\!]\varrho = \tag{3.5}$$

let

$$e'_1 = \mathcal{N}[\![\mathtt{e}_1]\!]\varrho \tag{1}$$

in

$$\mathcal{N}[\![\mathtt{e}]\!]\,\varrho[\mathtt{x} \mapsto e'_1] \tag{2}$$

end

B.2 The Online Abstract Interpreter

B.2.1 Constants

$$\mathcal{P}[\![\mathtt{const}]\!]\rho\,\delta\,\xi\,d = < \alpha(\mathtt{const}), \mathtt{const} > \tag{3.6}$$

B.2.2 Identifiers

$$\mathcal{P}[\![\mathtt{ident}]\!]\rho\,\delta\,\xi\,d \ = \ \rho(\mathtt{ident}) \tag{3.7}$$

B.2.3 Conditions

$$\mathcal{P}[\![(\texttt{if c } e_1\ e_2)]\!]\rho\,\delta\,\xi\,d = \tag{3.8}$$

let

$$< c^{\alpha}, c^{R} >= \mathcal{P}[\![c]\!]\rho\,\delta\,\xi\,d \tag{1}$$

$$< e^{\alpha}, e^{R} >= \begin{cases} \mathcal{P}[\![e_1]\!]\rho\,\delta\,\xi\,d & \text{if } \gamma(c^{\alpha}) = \texttt{true} \\ \mathcal{P}[\![e_2]\!]\rho\,\delta\,\xi\,d & \text{if } \gamma(c^{\alpha}) = \texttt{false} \\ < \perp, (\texttt{if c } e_1\ e_2) > & \text{if } \gamma(c^{\alpha}) = \perp \\ \mathcal{C}(c^{R}\ e_1\ e_2\ \rho\,\delta\,\xi) & \text{otherwise} \end{cases} \tag{2}$$

in

$$< e^{\alpha}, e^{R} > \tag{3}$$

end

$$\mathcal{C}(c^{R}\ e_1\ e_2\ \rho\,\delta\,\xi) = \tag{3.9}$$

let

$$< \rho_T, \rho_F >= \mathit{Split}(c^{R}, \rho) \tag{1}$$

$$< e_1^{\alpha}, e_1^{R} >= \mathcal{P}[\![e_1]\!]\rho_T\,\delta\,\xi\ \textbf{true} \tag{2}$$

$$< e_2^{\alpha}, e_2^{R} >= \mathcal{P}[\![e_2]\!]\rho_F\,\delta\,\xi\ \textbf{true} \tag{3}$$

in

$$< e_1^{\alpha}\ \nabla_P\ e_2^{\alpha},\ (\texttt{if } c^{R}\ e_1^{R}\ e_2^{R}\ ) > \tag{4}$$

end

B.2.4 Function Properties

Primitive operators

$$\mathcal{P}[\![(\mathrm{op}\ \mathrm{e}_1\,\mathrm{e}_2 \ldots \mathrm{e}_n)]\!]\rho\,\delta\,\xi\,d = \tag{3.12}$$

let

$$< \mathrm{e}_i^\alpha, \mathrm{e}_i^R >= \mathcal{P}[\![\mathrm{e}_i]\!]\rho\,\delta\,\xi\,d \quad \text{for all } 1 \le i \le n \tag{1}$$

$$v^\alpha = \mathit{apply}(\alpha(\mathrm{op}), \mathrm{e}_1^\alpha\,\mathrm{e}_2^\alpha \ldots \mathrm{e}_n^\alpha) \tag{2}$$

$$v^R = \begin{cases} \gamma(v^\alpha) & \text{if } \gamma(v^\alpha) \notin \{\top, \bot\} \\ (\mathrm{op}\ \mathrm{e}_1^R\,\mathrm{e}_2^R \ldots \mathrm{e}_n^R) & \text{otherwise} \end{cases} \tag{3}$$

in

$$< v^\alpha,\ v^R > \tag{4}$$

end

Function Application

$$\mathcal{P}[\![(\lambda \mathrm{x}.\mathrm{e})\ \mathrm{e}_1]\!]\rho\,\delta\,\xi\ \mathrm{false} = \tag{3.13}$$

let $< \mathrm{e}_1^\alpha, \mathrm{e}_1^R >= \mathcal{P}[\![\mathrm{e}_1]\!]\rho\,\delta\,\xi\ \mathrm{false}$ (1)

$$x^R = \begin{cases} \gamma(\mathrm{e}_1^\alpha) & \text{if } \gamma(\mathrm{e}_1^\alpha) \notin \{\top, \bot\} \\ \mathrm{x} & \text{otherwise} \end{cases} \tag{2}$$

$$< \mathrm{e}^\alpha,\ \mathrm{e}^R >= \mathcal{P}[\![\mathrm{e}]\!]\,\rho[\mathrm{x} \mapsto < \mathrm{e}_1^\alpha,\ x^R >]\,\delta\,\xi\ \mathrm{false} \tag{3}$$

$$v^R = \begin{cases} \gamma(\mathrm{e}^\alpha) & \text{if } \gamma(\mathrm{e}^\alpha) \notin \{\top, \bot\} \\ (\lambda \mathrm{x}.\mathrm{e}^R\ \mathrm{e}_1^R) & \text{otherwise} \end{cases} \tag{4}$$

in

$$< \mathrm{e}^\alpha,\ v^R > \tag{5}$$

end

Dynamic Function Application

$$\mathcal{P}[\![(\lambda x.\,e)e_1]\!]\rho\,\delta\,\xi\,\mathbf{true} = \tag{3.14}$$

let

$$< e_1^{\alpha}, e_1^{R} >= \mathcal{P}[\![e_1]\!]\rho\,\delta\,\xi\,\mathbf{true} \tag{1}$$

in

if $e_1^{\alpha} \sqsubseteq \delta(\lambda x.\,e_{id})$ **then** (2)

$$< \xi(\lambda x.\,e_{id}),\ (\lambda x.\,e\ e_1^{R}) > \tag{3}$$

else

let

$$\delta' = \delta[\lambda x.\,e_{id} \mapsto \delta(\lambda x.\,e_{id})\nabla_R\ e_1^{\alpha}] \tag{4}$$

$$x^R = \begin{cases} \gamma(\delta'(\lambda x.\,e_{id})) & \text{if } \gamma(\delta'(\lambda x.\,e_{id})) \notin \{\top, \bot\} \\ x & \text{otherwise} \end{cases} \tag{5}$$

$$\rho' = \rho[x \mapsto < \delta'(\lambda x.\,e_{id}), x^R >] \tag{6}$$

$$< e^{\alpha}, e^{R} >= \mathcal{P}[\![e]\!]\rho'\,\delta'\,\xi\,\mathbf{true} \tag{7}$$

$$v^R = \begin{cases} \gamma(e^{\alpha}) & \text{if } \gamma(e^{\alpha}) \notin \{\top, \bot\} \\ (\lambda x.\,e^{R}\ e_1^{R}) & \text{otherwise} \end{cases} \tag{8}$$

$$\xi' = \xi[\lambda x.\,e_{id} \mapsto \xi(\lambda x.\,e_{id})\nabla_R\ e^{\alpha}] \tag{9}$$

in

if $e^{\alpha} \sqsubseteq \xi(\lambda x.\,e_{id})$ **then** (10)

$$< e^{\alpha},\ v^R > \tag{11}$$

else

$$\mathcal{P}[\![(\lambda x.\,e)e_1]\!]\rho\,\delta\,\xi'\,\mathbf{true} \tag{12}$$

end

end

Bibliography

[1] *Programming Languages — C*, 1990. ANSI/ISO 9899-1990.

[2] ABRAMSKY, S., AND HANKIN, C., Eds. *Abstract Interpretation of Declarative Languages*. Chichester: Ellis Horwood, 1987.

[3] AHO, A., SETHI, R., AND ULLMAN, J. *Compilers. Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[4] ALLISON, L. *A Practical Introduction to Denotational Semantics*. Cambridge: Cambridge University Press, 1989.

[5] ANDERSEN, L. Self-applicable C program specialization. In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992 (Technical Report YALEU/DCS/RR-909)* (June 1992), New Haven, CT: Yale University, pp. 54–61.

[6] ANDERSEN, L. Binding-time analysis and the taming of C pointers. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993* (1993), New York: ACM, pp. 47–58.

[7] ANDERSEN, L. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1994. DIKU Research Report 94/19.

[8] ANDERSEN, P. Partial evaluation applied to ray tracing. DIKU Research Report 95/2, DIKU, University of Copenhagen, Denmark, 1995.

[9] BAIER, R., GLÜCK, R., AND ZÖCHLING, R. Partial evaluation of numerical programs in Fortran. In *Partial Evaluation and Semantics-Based Program Ma-*

nipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne) (1994), pp. 119–132.

[10] BJØRNER, D., ERSHOV, A., AND JONES, N., Eds. *Workshop Compendium, Workshop on Partial Evaluation and Mixed Computation, Gl. Avernæs, Denmark, October 1987*. Department of Computer Science, Technical University of Denmark, Lyngby, Denmark, 1987.

[11] BONDORF, A. Automatic autoprojection of higher order recursive equations. In *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990 (Lecture Notes in Computer Science, vol. 432)* (May 1990), N. Jones, Ed., Berlin: Springer-Verlag, pp. 70–87. Revised version in [12].

[12] BONDORF, A. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming 17* (1991), 3–34.

[13] BONDORF, A., AND MOGENSEN, T. Logimix: A self-applicable partial evaluator for Prolog. DIKU, University of Copenhagen, Denmark, May 1990.

[14] BOURNDONCLE, F. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming 2*, 4 (October 1992), 407–435.

[15] BUHR, P. A., MACDONALD, H. I., AND STROOBOSSCHER, R. A. μSystem annotated reference manual, version 4.4.3. Tech. Rep. Unnumbered (Available via `ftp` to `plg.uwaterloo.ca` in `pub/uSystem/uSystem.ps.Z`.), Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, Mar. 1991.

[16] BULYONKOV, M., AND ERSHOV, A. How do ad-hoc compiler constructs appear in universal mixed computation processes? In *Partial Evaluation and Mixed Computation* (1988), D. Bjørner, A. Ershov, and N. Jones, Eds., Amsterdam: North-Holland, pp. 65–81.

[17] CHAMBERS, C. The design and implementation of the Self compiler, an optimizing compiler for object-oriented programming languages. Tech. Rep. STAN-CS-92-1420, Stanford, 1992.

[18] CHAMBERS, C. Object-oriented multi-methods in Cecil. *ECOOP '92 Conference Proceedings* (July 1992).

[19] CLINGER, W., AND REES, J. Revised(4) report on the algorithmic language Scheme. *ACM Lisp Pointers IV* (July—Sept. 1991).

[20] COLBY, C., AND LEE, P. A modular implementation of partial evaluation. Tech. Rep. CMU-CS-92-123, School of Computer Science, Carnegie Mellon University, March 1992.

[21] CONSEL, C. Binding time analysis for higher order untyped functional languages. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France* (1990), New York: ACM, pp. 264–272.

[22] CONSEL, C. Polyvariant binding-time analysis for applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993* (1993), New York: ACM, pp. 66–77.

[23] CONSEL, C., AND DANVY, O. For a better support of static data flow. In *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)* (1991), J. Hughes, Ed., ACM, Berlin: Springer-Verlag, pp. 496–519.

[24] CONSEL, C., AND KHOO, S. On-line and off-line partial evaluation: Semantic specifications and correctness proofs. Tech. Rep. YALE-DCS-tr912, Yale, 1993. To appear in *Journal of Functional Programming*.

[25] CONSEL, C., AND KHOO, S. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems 15*, 3 (1993), 463–493.

[26] CONSEL, C., AND PAI, S. A programming environment for binding-time based partial evaluators. In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992 (Technical Report YALEU/DCS/RR-909)* (1992), New Haven, CT: Yale University, pp. 62–66.

[27] COUSOT, P., AND COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *ACM Symposium on Principles of Programming Languages* (January 1977), 238–252.

[28] CYTRON, R., FERRANTE, J., ROSEN, B., WEGMAN, M., AND ZADECK, F. K. Efficiently computing static single assignment form and the control dependence

graph. *ACM Transactions on Programming Languages and Systems 13*, 4 (Oct. 1991), 451–490.

[29] CYTRON, R., AND GERSHBEIN, R. Efficient accomodation of may-alias information in SSA form. *SIGPLAN Notices 28*, 6 (June 1993), 36–45. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[30] DAVEY, B., AND PRIESTLEY, H. *Introduction to Lattices and Order.* Cambridge Press, 1990.

[31] DEAN, J., CHAMBERS, C., AND GROVE, D. Identifying profitable specialization in object-oriented languages. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)* (1994), pp. 85–96.

[32] ERSHOV, A. A theoretical principle of system programming. *Soviet Mathematics Doklady 18*, 2 (1977), 312–315.

[33] FEGARAS, L., SHEARD, T., AND ZHOU, T. Improving programs which recurse over multiple inductive structures. *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)* (1994), 21–32.

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

[35] GOMARD, C., AND JONES, N. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming 1*, 1 (January 1991), 21–69.

[36] HAO, J.-K., AND CHABRIER, J.-J. Combining partial evaluation and constraint solving: a new approach to constraint logic programming. In *2nd International IEEE Conference on Tools for Artificial Intelligence, Herndon, VA, USA* (1990), New York: IEEE Computer Society, pp. 494–500.

[37] HARBISON, S. *Modula-3.* Englewood Cliffs, NJ: Prentice Hall, 1992.

[38] HARNETT, S., AND MONTENYOHL, M. Towards efficient compilation of a dynamic object-oriented language. In *Partial Evaluation and Semantics-Based*

Program Manipulation, San Francisco, California, June 1992 (Technical Report YALEU/DCS/RR-909) (1992), New Haven, CT: Yale University, pp. 82–89.

[39] HENDREN, L. Parallelizing programs with recursive data structures. Tech. Rep. 90-1114, Cornell University, 1990. (Ph.D. Thesis Chapters 3 and 4 – Interference Analysis).

[40] HENDREN, L., AND NICOLAU, A. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems 1*, 1 (January 1990), 34–47.

[41] HENGLEIN, F. 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)* (1991), J. Hughes, Ed., ACM, Berlin: Springer-Verlag, pp. 448–472.

[42] HENGLEIN, F., AND MOSSIN, C. Polymorphic binding-time analysis. In *Programming Languages and Systems — ESOP'94. 5th European Symposium on Programming, Edinburgh, U.K., April 1994 (Lecture Notes in Computer Science, vol. 788)* (1994), D. Sannella, Ed., Berlin: Springer-Verlag, pp. 287–301.

[43] HUGHES, J. Backwards analysis of functional programs. In *Partial Evaluation and Mixed Computation* (1988), D. Bjørner, A. Ershov, and N. Jones, Eds., Amsterdam: North-Holland, pp. 187–208.

[44] JONES, M. Dictionary-free overloading by partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)* (1994), pp. 107–117.

[45] JONES, N. Flow analysis of lazy higher-order functional programs. In *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin, Eds. Chichester: Ellis Horwood, 1987, pp. 103–122.

[46] JONES, N., GOMARD, C., AND SESTOFT, P. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[47] JONES, N., AND MYCROFT, A. Data flow analysis of applicative programs using minimal function graphs. In *Thirteenth ACM Symposium on Principles of Pro-*

gramming Languages, St. Petersburg, Florida. New York: ACM, 1986, pp. 296–306.

[48] JONES, N., SESTOFT, P., AND SØNDERGAARD, H. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation 2*, 1 (1989), 9–50.

[49] KARR, M. Code generation by coagulation. In *Conference Record of the 1984 ACM SIGPLAN Symposium on Compiler Construction* (June 1984), vol. 19, Association for Computing Machinery, pp. 1–12.

[50] KHOO, S., AND SUNDARESH, R. Compiling inheritance using partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (Sigplan Notices, vol. 26, no. 9, September 1991)* (1991), New York: ACM, pp. 211–222.

[51] LAKHOTIA, A., AND STERLING, L. ProMiX: A Prolog partial evaluation system. In *The Practice of Prolog*, L. Sterling, Ed. Cambridge, MA: MIT Press, 1991, ch. 5, pp. 137–179.

[52] LAM, J., AND KUSALIK, A. A partial evaluation of partial evaluators for pure Prolog. Tech. Rep. TR 90-9, Department of Computational Science, University of Saskatchewan, Canada, November 1990.

[53] LANDI, W., AND RYDER, B. G. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Notices 27*, 7 (July 1992), 235–248. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[54] LAUNCHBURY, J. *Projection Factorisations in Partial Evaluation*. Cambridge: Cambridge University Press, 1991.

[55] LAWALL, J. Proofs by structural induction using partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993* (1993), New York: ACM, pp. 155–166.

[56] LEONE, M., AND LEE, P. Lightweight run-time code generation. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)* (1994), pp. 97–106.

[57] LIU, Y., STOLLER, S., AND TEITELBAUM, T. Discovering auxiliary information for incremental computation. *ACM Symposium on Principles of Programming Languages* (January 1996).

[58] MALMKJÆR, K., HEINTZE, N., AND DANVY, O. ML partial evaluation using set-based analysis. In *1994 ACM SIGPLAN Workshop on ML and Its Applications, Orlando, Florida, June 1994 (Technical Report 2265, INRIA Rocquencourt, France)* (1994), pp. 112–119.

[59] MARQUARD, M., AND STEENSGAARD, B. Partial evaluation of an object-oriented imperative language. Master's thesis, DIKU, University of Copenhagen, Denmark, April 1992. Available from `ftp.diku.dk` as file `pub/diku/semantics/papers/D-152.ps.Z`.

[60] MASON, D. V. A Functional intermediate from for diverse source languages. Submitted to CASCON 1996.

[61] MEYER, U. 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)* (1991), New York: ACM, pp. 94–105.

[62] MEYER, U. Correctness of online partial evaluation for a pascal-like language. Bericht 9205, AG Informatik, Universität Giessen, Germany, 1992.

[63] MILNER, R. *The Definition of Standard ML.* Cambridge, MA: MIT Press, 1990.

[64] MILNER, R., TOFTE, M., AND HARPER, R. *The definition of Standard ML.* MIT Press, Cambridge, Mass., 1990.

[65] MOGENSEN, T. The application of partial evaluation to ray-tracing. Master's thesis, DIKU, University of Copenhagen, Denmark, 1986.

[66] MOGENSEN, T. Partially static structures in a self-applicable partial evaluator. In *Partial Evaluation and Mixed Computation* (1988), D. Bjørner, A. Ershov, and N. Jones, Eds., Amsterdam: North-Holland, pp. 325–347.

[67] MOGENSEN, T., AND BONDORF, A. Logimix: A self-applicable partial evaluator for Prolog. In *LOPSTR 92. Workshops in Computing* (Jan. 1993), K.-K. Lau and T. Clement, Eds., Berlin: Springer-Verlag.

[68] NIRKHE, V., AND PUGH, W. 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* (1992), New York: ACM, pp. 269–280.

[69] PARRAIN, A., DEVIENNE, P., AND LEBEGUE, P. Towards optimization of full Prolog programs guided by abstract interpretation. In *WSA '92, Static Analysis, Bordeaux, France, September 1992. Bigre vols 81–82, 1992* (1992), M. Billaud et al., Eds., Rennes: IRISA, pp. 295–303.

[70] RUF, E. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, California, February 1993. Published as technical report CSL-TR-93-563.

[71] RUF, E., AND WEISE, D. Avoiding redundant specialization during partial evaluation. Tech. Rep. CSL-TR-92-518, Computer Systems Laboratory, Stanford University, Stanford, CA, April 1992.

[72] RUF, E., AND WEISE, D. Improving the accuracy of higher-order specialization using control flow analysis. In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992 (Technical Report YALEU/DCS/RR-909)* (1992), New Haven, CT: Yale University, pp. 67–74.

[73] RYTZ, B., AND GENGLER, M. A polyvariant binding time analysis. In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992 (Technical Report YALEU/DCS/RR-909)* (1992), New Haven, CT: Yale University, pp. 21–28.

[74] SAKAMA, C., AND ITOH, H. Partial evaluation of queries in deductive databases. *New Generation Computing 6*, 2,3 (1988), 249–258.

[75] SHIVERS, O. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991.

[76] SMITH, D. Partial evaluation of pattern matching in constraint logic programming languages. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (Sigplan Notices, vol. 26, no. 9, September 1991)* (1991), New York: ACM, pp. 62–71.

[77] SRIDHAR, A., AND VEMURI, R. Automatic precondition verification for high-level design transformations. In *1990 IEEE International Symposium on Circuits and Systems* (1990), New York: IEEE, pp. 2654–2657.

[78] STEELE JR., G. *Common Lisp Lanuguage 2nd Revised Ed.* Englewood Cliffs, NJ: Prentice Hall, 1989.

[79] STROUSTRUP, B. *The C++ Programming Language*, second ed. Reading, MA: Addison-Wesley, 1993.

[80] UNITED STATES DEPARTMENT OF DEFENSE. *Reference Manual for the ADA Programming Language*, 1983. ANSI/MIL-STD-1815A-1983.

[81] VASELL, J. A partial evaluator for data flow graphs. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993* (1993), New York: ACM, pp. 206–215.

[82] WEGMAN, M., AND ZADECK, F. K. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems 13*, 2 (April 1991), 181–210.

[83] WEISE, D., CONYBEARE, R., RUF, E., AND SELIGMAN, S. Automatic online partial evaluation. In *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)* (1991), J. Hughes, Ed., Berlin: Springer-Verlag, pp. 165–191.

[84] ZAKHAROVA, N., PETRUSHIN, V., AND YUSHCHENKO, E. Denotational semantics of mixed computation processes. In *[10]* (1987), pp. 379–388.