Partial Evaluation for Program Analysis

Daniel Damian

Progress Report

(revised version)

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

June 1998 **Supervisor:** Olivier Danvy

Preface

This report documents the scientific work carried out during the part A of the Ph.D. studies of author at BRICS[1]. The main topic of interest is programming languages, focusing on partial evaluation and program transformation.

One of the ideas explored in the period was to assess the impact of partial evaluation on program analysis. The present report makes an account of the results obtained, from both theoretical and practical points of view.

The report presents a case study of partial evaluation in program analysis. Starting from the observation that abstract interpretation incurs an interpretive overhead in the same way as a standard interpretation, we explore the benefits that can be obtained from specializing an abstract interpretation-based program analysis with respect to the source program.

The contribution of our work is to present a complete formal account of Shivers's 0-CFA implementation and specialization. In the first chapters of the report, we point out a missing link between the formal specification of Shivers's 0-CFA and its implementation. We fill this gap by formally relating the specification with the formalization of the implementation.

Both implementation and specialization are formalized within a unified programming-language framework. We formally specify an implementation language, thus making it possible to prove correctness of both implementation and specialization. The former is done by formally relating the semantics of the program implementing the analysis to the specification of the analysis. The later is done by establishing meaning-preserving program transformations which lead to program specialization. We thus obtain a correct specialized analyzer by means of simple program transformations.

The motivation of this work is to build a complete theoretical foundation of an idea which can prove useful in practice. Our approach constructs a formal connection between a mathematical solution to a problem, its practical implementation, and the improvement obtained by specialization.

Acknowledgments I would like to thank my supervisor Olivier Danvy for the numerous discussions and for his valuable comments about my work. I am also grateful to Andrzej Filinski for his courses at DAIMI and for his numerous suggestions.

Thanks are also due to Dominique Boucher, who generously provided the source of his benchmarks. I would also like to thank Torben Amtoft for his collaboration during his stay at BRICS. Finally I thank Bernd Grobauer and Zhe Yang for both their help and enjoyable company.

[1] Basic Research In Computer Science (`http://www.brics.dk`), Centre of the Danish National Research Foundation.

Contents

List of Figures

Chapter 1

Introduction

Program analysis is the mainstay of today's compilers. Program analysis builds a bridge between expressiveness of programming languages and efficiency of implementation. As program optimization techniques rely on assumptions about the behavior of the program, program analysis provides methods to automatically validate such assumptions.

Program analysis computes an approximation of the run-time behavior of a program, with the purpose of certifying automatic transformation of this program. There is always a trade-off between the precision of the approximation and the speed of the analyzing process, as the more precise information we want to obtain about the program, the more time the has to be spend to compute this information.

On the other end, partial evaluation is a general technique for optimizing programs, which attempts to remove redundant computation. In both of its forms — online or offline — [4, 9] partial evaluation is a process of specializing a program P with respect to part of its input. It is most effective when it involves removing interpretive overhead incurred by repetitive traversal of data structures.

Static program analysis is useful in offline partial evaluation [4, 9], as for instance binding-time analysis or closure analysis are extensively employed. The converse can be also true. As program analysis often involves iterating a computation over the syntax of the program, we can identify an interpretative overhead.

Our goal is to eliminate the interpretive overhead of program analysis over a given program, using partial evaluation. In our case, we specialize a program analyzer PA with respect to the program to analyze `p`. In our case having:

$$\mathbf{run}\ PA\ \langle \texttt{p}, \texttt{init-store} \rangle = r$$

we to produce $PA_{\texttt{p}}$ such that:

$$\mathbf{run}\ PA_{\texttt{p}}\ \langle \texttt{init-store} \rangle = r$$

In this work we make a case study of using partial evaluation in program analysis. As an instance of program analysis, control flow analysis (CFA) computes what functions occur at application points of the program. Our specialization target is the 0-CFA version of Shivers's more general k-CFA, as pioneered in his Ph.D. thesis [14].

Our study focuses on two issues. The first result we obtain is a provably correct efficient implementation of Shivers's 0-CFA.

In Chapter 5 of his thesis, Shivers points out that a faithful implementation of his formal control-flow semantics involves prohibitively expensive memory operations. Shivers obtains an efficient implementation by sequentializing recursive calls and maintaining just one global copy of a variable environment, thus obtaining a less precise, but still safe analysis.

Shivers justifies the method relying on the monotonicity of the control-flow semantics, using an algorithm for computing the transitive closure of a function. In our view, there is still a gap between his argument and the actual implementation, since the former relies on properties of the formal control-flow semantics, which do not correspond with the semantics of the implementation.

Instead, we are reformulating the 0-CFA analysis in a new set of equations that compute the result obtained from the implementation. We prove that the modified 0-CFA is safe by formally relating it with Shivers's original analysis.

To formalize the implementation of the analysis and its specialization, we take a uniform programming-language approach. We formalize a a subset of the actual implementation language, sufficient for programming the analysis and its specialization.

The implementation language is formalized by specifying its syntax and formal semantics, in the form of denotational semantics. The implementation is then a program in this language. The correctness of implementation is proved by showing that the meaning of the program corresponds to the result obtained by the analysis.

Using the semantics of the implementation language, we establish several generally valid program transformations that lead to specialization. We produce a specialized analyzer by means of such program transformations, thus obtaining its correctness.

We also account for the practical impact of our method. We present the experimental results obtained comparing the running times of the standard and specialized analysis over a set of benchmarks.

1.1 Related Work

In a previous work by Boucher and Feeley [3], a similar idea is considered. The idea presented in their work is that since from an interpreter one can obtain a compiler by partial evaluation, from an abstract interpreter one can obtain an "abstract compiler" by specialization as well.

Instead of a traditional partial evaluation approach of producing residual code, their rezidualized program is in the form of closures constructed in the memory, similarly to Feeley's Scheme compiler [6, 7]. Boucher and Feeley also point out that the approach does not completely remove the interpretation overhead, since closures themselves are repeatedly traversed when the residual program is run. This approach is taken as a tradeoff of efficiency to avoid compilation of residual code.

Also, the 0-CFA algorithm they are using is a non-formalized compositional variant of Shivers's analysis, which, as we have observed, although more suitable to specialization, is more conservative.

Among other methods of computing the control flow analysis for higher-order languages we mention a method based on constraints, initially formulated for type inference in object-oriented languages [11] and then for higher-order functional languages in [12, 8]. The idea of computing the control flow by constructing a set of constraints from the target program might seem to be a way of removing the syntactic overhead.

The issue is more extensively dealt in an ongoing work, together with Torben Amtoft and Olivier Danvy, which, in a similar fashion as the present work, attempts to assess the benefits obtained by removing the constraint solving overhead by specialization.

1.2 Prerequisites

We expect the reader to be familiar with Shivers's 0-CFA described in [14]. The analysis computes an approximative solution to the control flow problem in a higher-order, untyped functional language: given a closed program, find what functions are called at each application point in the program in any run of the program. The source programs are assumed to be in Continuation Passing Style [13, 15].

1.3 Notations

In the following presentation we make use of basic notions of domain theory, as can be found in Winskel's textbook [16]. A cpo is a set together with a complete partial order relation. For A and B cpo's, $A \to_c B$ is the cpo of continuous functions from A to B. The cpo $A_\perp$ is the lifted counterpart of A, for which we use $\mathtt{up} : A \to A_\perp$ as the lifting function. We use $f^\dagger : A_\perp \to B_\perp$ as the strict extension of a function $f : A \to B_\perp$. For a continuous function $f : A \to A$ on a pointed domain A, $Fix(f)$ denotes its least fixed point.

Given a function $f : A \to B$, $x \in A$ and $y \in B$ we use $f[x \mapsto y]$ for the function defined as y in x and as f everywhere else. We consider $[x \mapsto y]$ as $\perp[x \mapsto y]$, where applicable. For a set of points x_i and values y_i we use $[x_i \mapsto y_i]_i$ as a short notation for $[x_1 \mapsto y_1]...[x_n \mapsto y_n]$ where n can be deduced from the context.

We use A^* for the set of finite vectors of elements from the set A and its elements are written as $\langle x_1, ..., x_n \rangle$. For tuples, we use π_i to denote the i-th projection.

At the meta-level, we use **if** p **then** v_1 **else** v_2 as a shorthand for the value defined as v_1 if p holds or v_2 otherwise.

1.4 Overview

In Chapter 2 we recall the 0-CFA as defined by Shivers. We start by giving the syntax of the CPS subset of Scheme (Figure 2.1) which is the target language of the analysis, and we define the domains used in further developments. In Section 2.3 we recall the formal definition of the 0-CFA and we conclude by stating a tension between the formal 0-CFA semantics and the implementation method suggested by Shivers.

In Chapter 3 we account for the implementation aspects of the 0-CFA by presenting a modified version of the analysis. In Section 3.3 we prove the correctness of the modified analysis by establishing a relationship with the one defined by Shivers.

In Chapter 4 we present a programming language for implementing the analysis from Chapter 3, which will also be used for formalizing the specialization process. In Section 4.3 we give formal semantics for our language in the form of denotational semantics. In Section 4.4 we implement the 0-CFA as defined in Chapter 3 in our new language, and we prove the correctness of the implementation.

In Chapter 5 we present a notion of program equivalence and we enumerate several program transformations preserving equivalence which are used in specialization.

In Chapter 6 we specialize the program from Section 4.4 by performing a series of transformations, using the equivalence results established in the previous chapter. We continue by observing more specialization opportunities, specific to the program implementing the analysis. More transformations are performed and proven correct.

In Chapter 7 we assess the practical impact of our approach. We present the results we have obtained by comparing the running times experienced for both implementations of the analysis over a set of benchmarks.

We conclude in Chapter 8 and comment on further work and perspectives.

Chapter 2

Shivers's Specification of the 0-CFA

2.1 Introduction

In his thesis [14], Olin Shivers introduces a notion of k-CFA. Starting from standard interpreting semantics for CPS-transformed programs, Shivers derives control-flow semantics by means of abstract interpretation [5].

In this chapter we recall the definition of the analysis. We present the source language of the analysis and the domains induced by the source program, which will also be used in throughout this report. We present the 0-CFA form of the formal control-flow equations stated by Shivers. We conclude by making several remarks about the tension between the specification of the analysis and the implementation method proposed in Shivers's thesis.

2.2 The Source Language

The source language of the analysis, displayed in Figure 2.1, is a CPS subset of the Scheme language [10]. The CPS subset is considered as an intermediate language for a Scheme compiler. Shivers shows in his thesis how whole Scheme programs can be translated into the reduced subset.

Similarly to Shivers's treatment, for brevity, we only account for a couple of primitive operators in the language. Our development can be easily extended to all the other primitive operators. The primitive operators are also in CPS, i.e., they receive an extra parameter – the continuation. We restrict ourself to programs without side-effects, although the result of this work can be also be extended to handle such cases, in a similar way with the one described in Shivers's thesis.

The source program *Prg* induces several finite sets of terms and labels, defined in Figure 2.2. We should say that *LAB* contains all the labels in *Prg*, and all the labels of call sites inside primary operators. For instance, for an existing p:`if`, we have two additional labels $\texttt{if}_p^t$ and $\texttt{if}_p^f$ for the call sites of the two branches of the conditional, and similarly we have a label $\texttt{+}_p$ for an operator p:`+`.

We define a constant *Stop* to denote the top-level continuation, and, for uniformity, we introduce *LStop* as its label. We use $\ulcorner l \urcorner$ to denote the term labeled by label $l \in LAB$.

We consider $\Psi = |PROC|$ and $\{\texttt{l}_1, ..., \texttt{l}_\Psi\}$ to be the elements of *PROC*. We define Θ as the maximum arity of the procedures in *Prg*. We use the function *arity* : $PROC \to \mathbf{N}$ to obtain the arity of a procedure in *Prg*.

$$\begin{array}{rcl}
Prg, d \in Lam & ::= & l\texttt{:(}\lambda\ \texttt{(}id_1\ \ldots\ id_n\texttt{)}\ e\texttt{)} \\
e \in Call & ::= & c\texttt{:(}f\ a_1\ \ldots\ a_n\texttt{)} \\
 & | & c\texttt{:(letrec ((}x_1\ d_1\texttt{)}\ldots\texttt{)}\ e\texttt{)} \\
f \in Fun & = & Lam + Var + Prim \\
a \in Arg & = & Lam + Var + Const \\
x \in Var & = & \{\texttt{x}, \ldots\} \\
k \in Const & = & \{\texttt{0}, \texttt{1}, \texttt{\#t}, \ldots\} \\
Prim & = & \{p\texttt{:+}, p\texttt{:if}\} \\
l, c, p \in Lab & &
\end{array}$$

Figure 2.1: Abstract syntax of labeled CPS terms

VAR :	the set of variables in *Prg*
LAB :	the set of labels in *Prg*
PRIM :	the set of *Prim* terms in *Prg*
LAM :	the set of *Lam* terms in *Prg*
CALL :	the set of *Call* terms of *Prg*
ARG :	the set of *Arg* terms of *Prg*
FUN :	the set of *Fun* terms of *Prg*
PROC :	$LAM + PRIM + \{Stop\}$
LPROC :	the labels of the terms in *PROC*

Figure 2.2: Sets induced by the source program

2.3 Specification of Shivers's 0-CFA

The 0-CFA analysis as defined by Shivers, together with domain definitions and functionalities, is recalled in Figure 2.3. The domains of definition are considered here as complete lattices, with the natural orderings on powersets and functions. For readability, tuples appearing at the argument position are written in curried form.

The analysis is computed by two mutually recursive functions. Function $\widehat{\mathcal{C}}$ propagates control-flow information at an application site, and function $\widehat{\mathcal{F}}$ simulates the "unfolding" of an application. The result of the analysis of a source program *Prg* is a call cache, i.e., a map assigning to each call site $c \in CALL$ a set $A \in \mathcal{P}(PROC)$ of procedures that may be called at that site in an execution of the program *Prg*. The variable environment $\widehat{\rho}$ maps each variable $v \in VAR$ to a similar set of procedures as possible values in a run of *Prg*. The function $\widehat{\mathcal{A}}$ is used to evaluate arguments in a given environment.

The functions $\widehat{\mathcal{C}}$ and $\widehat{\mathcal{F}}$ are taken as elements of the domains *CSPACE* and *FSPACE* defined as follows:

$$\begin{array}{llll}
CState & = (CALL \times \widehat{Venv}) & CSPACE & = CState \to \widehat{CCache} \\
FState & = (PROC \times (\mathcal{P}(PROC))^* \times \widehat{Venv}) & FSPACE & = FState \to \widehat{CCache}
\end{array}$$

As it is only necessary to define $\widehat{\mathcal{F}}$ on a subset of *FState*, it is considered as restricted by the formula:

$$(l, \langle A_1, \ldots, A_n\rangle, \widehat{\rho}) \in FState \iff arity(l) = n$$

The functions $\widehat{\mathcal{C}}$ and $\widehat{\mathcal{F}}$ are defined as the least fixed point of an operator $H : (CSPACE \times FSPACE) \to (CSPACE \times FSPACE)$. For an arbitrary pair $\big(\widehat{\mathcal{C}}, \widehat{\mathcal{F}}\big)$, $H\big(\widehat{\mathcal{C}}, \widehat{\mathcal{F}}\big)$ is obtained from the recursive equations in Figure 2.3, by using $\widehat{\mathcal{C}}$ and $\widehat{\mathcal{F}}$ on the right side of the equations.

Shivers proves that H is a well-defined operator, it is monotone, continuous and it has a least fixed point. The least fixed point is obtained by constructing a chain of approximations defined

$$\begin{array}{rcll} \widehat{CCache} & = & LAB \to \mathcal{P}(PROC) & \text{– The call cache} \\ \widehat{Venv} & = & VAR \to \mathcal{P}(PROC) & \text{– The variable environment} \end{array}$$

$$\begin{array}{rcl} \widehat{\mathcal{A}} & : & (ARG \cup FUN) \to \widehat{Venv} \to \mathcal{P}(PROC) \\ \widehat{\mathcal{C}} & : & (CALL \times \widehat{Venv}) \to \widehat{CCache} \\ \widehat{\mathcal{F}} & : & (PROC \times (\mathcal{P}(PROC))^* \times \widehat{Venv}) \to \widehat{CCache} \end{array}$$

$$\begin{array}{rcl} \widehat{\mathcal{A}}[\![k]\!]\widehat{\rho} & = & \emptyset \\ \widehat{\mathcal{A}}[\![p{:}\texttt{if}]\!]\widehat{\rho} & = & \{p{:}\texttt{if}\} \\ \widehat{\mathcal{A}}[\![l{:}\lambda]\!]\widehat{\rho} & = & \{l{:}\lambda\} \\ \widehat{\mathcal{A}}[\![x]\!]\widehat{\rho} & = & \widehat{\rho}(x) \end{array}$$

$$\begin{array}{rcl} \widehat{\mathcal{C}}\ [\![c{:}\texttt{(}f\ a_1\ ...\ a_n\texttt{)}]\!]\ \widehat{\rho} & = & \left(\bigsqcup_{f\in F} \widehat{\mathcal{F}}\ f\ \langle A_1, ..., A_n\rangle\ \widehat{\rho}\right) \sqcup [c \mapsto F] \\ & & \text{where } F = \widehat{\mathcal{A}}[\![f]\!]\widehat{\rho}, \quad A_i = \widehat{\mathcal{A}}[\![a_i]\!]\widehat{\rho}, \\ \widehat{\mathcal{C}}\ [\![c{:}\texttt{(letrec ((}x_1\ d_1\texttt{)}...\texttt{(}x_n\ d_n\texttt{))}\ e\texttt{)}]\!]\ \widehat{\rho} & = & \widehat{\mathcal{C}}\ [\![e]\!]\ \left(\widehat{\rho} \sqcup \bigsqcup_{i=1}^{n} [x_i \mapsto \widehat{\mathcal{A}}[\![d_i]\!]\widehat{\rho}]\right) \end{array}$$

$$\begin{array}{rcl} \widehat{\mathcal{F}}\ [\![l{:}\texttt{(}\lambda\ \texttt{(}x_1\ ...\ x_n\texttt{)}\ e\texttt{)}]\!]\ \langle A_1, ..., A_n\rangle\ \widehat{\rho} & = & \widehat{\mathcal{C}}\ [\![e]\!]\ \left(\widehat{\rho} \sqcup \bigsqcup_{i=1}^{n} [x_i \mapsto A_i]\right) \\ \widehat{\mathcal{F}}\ [\![p{:}\texttt{if}]\!]\ \langle B, K_1, K_2\rangle\ \widehat{\rho} & = & \left(\bigsqcup_{f\in K_1} \widehat{\mathcal{F}}\ f\ \langle\rangle\ \widehat{\rho}\right) \sqcup \left(\bigsqcup_{f\in K_2} \widehat{\mathcal{F}}\ f\ \langle\rangle\ \widehat{\rho}\right) \\ & & \sqcup[\texttt{if}_p^t \mapsto K_1, \texttt{if}_p^f \mapsto K_2] \\ \widehat{\mathcal{F}}\ [\![p{:}\texttt{+}]\!]\ \langle X, Y, K\rangle\ \widehat{\rho} & = & \left(\bigsqcup_{f\in K} \widehat{\mathcal{F}}\ f\ \langle\emptyset\rangle\ \widehat{\rho}\right) \sqcup [\texttt{+}_p \mapsto K] \\ \widehat{\mathcal{F}}\ Stop\ \langle A_1, ..., A_n\rangle\ \widehat{\rho} & = & \emptyset \end{array}$$

Figure 2.3: Shivers's original 0-CFA

by successive applications of H to the bottom value $(\lambda c.\emptyset, \lambda f.\emptyset)$. Let $(\widehat{\mathcal{C}}, \widehat{\mathcal{F}}) = H(\widehat{\mathcal{C}}, \widehat{\mathcal{F}})$ to be the limit of this chain.

Considering the source program *Prg* as an element of $PROC$, the 0-CFA analysis of *Prg* is given by

$$\widehat{\mathcal{F}}\ Prg\ \langle\{Stop\}\rangle\ \emptyset$$

2.4 Summary and Conclusions

In this chapter we have presented the 0-CFA analysis as defined by Shivers.

However, direct implementation of the 0-CFA semantic equations in inefficient. When we recursively call $\widehat{\mathcal{F}}$ over a set of procedures, we need to make copies of the variable environment, which is an expensive operation. In his thesis, Shivers informally describes a method to overcome the problem. Because of the monotonicity of the semantic functions, keeping just one copy of the variable environment and sequentializing the recursive calls produces a less precise analysis, with the benefit of reducing the memory usage. A technique of time stamps is suggested to ensure the termination of the computation.

However, by implementing this method, we are in fact implementing a different specification of the analysis. In the next chapters we modify the formulation of the analysis in order to correspond to the result computed by the actual implementation. We prove that the analysis obtained is safe with respect to Shivers's original, thus enabling us to prove the correctness of implementation.

Chapter 3

Specification of Implementation of 0-CFA

3.1 Introduction

In this chapter we formalize Shivers's informal description of an efficient implementation of the 0-CFA. We redesign the abstract semantics, in a way that fully accounts for the implementation issues by introducing the following changes:

– We consider a state-passing abstract 0-CFA semantics where a single copy of the variable environment $\widehat{\rho}$ and of the call cache $\widehat{\delta}$ is needed at one time, by sequentializing the recursive calls to the $\widehat{\mathcal{F}}$ function over a set of procedures.

– We introduce a timer t which is a "stamp" of the current environment. The timer is incremented each time the variable environment is increased. For each call site c, calls to the $\widehat{\mathcal{C}}$ function on c are monitored by a table $\widehat{\tau}$. The table keeps track of the value of the timer at the last (time-wise) call to $\widehat{\mathcal{C}}$.

– We consider a sequential update of the variable environment by updating one variable at the time, instead of a parallel update.

– We consider the variable environment as well as the call cache to be defined in terms of labels instead terms.

We provide a formal proof of correctness for the modified version of the analysis, by proving that the result returned is safe with respect to Shivers's analysis from Chapter 2.

3.2 State-passing 0-CFA

The domains and signatures of the modified 0-CFA semantics can be found in Figure 3.1. The variable environment, the call cache, the time stamps table and the timer are tupled together into a sequentially updated state variable. Along with redefining $\widehat{\mathcal{A}}$, $\widehat{\mathcal{F}}$ and $\widehat{\mathcal{C}}$ functions, we define additional functions:

- *Acc* is sequentializing the recursive calls to $\widehat{\mathcal{F}}'$ over a set of labels.
- *Union* is updating the variable environment, increasing the timer, if necessary.
- *UnionN* is sequentializing a parallel update of the variable environment, updating one variable at the time.

$$
\begin{array}{rcll}
\widehat{\delta} \in \widehat{CCache'} & = & LAB \to \mathcal{P}(LPROC) & \text{– The call cache} \\
\widehat{\rho} \in \widehat{Venv'} & = & VAR \to \mathcal{P}(LPROC) & \text{– The variable environment} \\
\widehat{\tau} \in Times & = & CALL \to \mathbf{N} & \text{– The time-stamps table} \\
\sigma \in Store & = & \widehat{Venv'} \times \widehat{CCache'} \times Times \times \mathbf{N} & \text{– The store}
\end{array}
$$

$$
\begin{array}{rcl}
\widehat{\mathcal{A}}' & : & (ARG \cup FUN) \to Store' \to \mathcal{P}(LPROC) \\
\widehat{\mathcal{C}}' & : & CALL \to Store' \to Store' \\
\widehat{\mathcal{F}}' & : & (PROC \times \mathcal{P}(LPROC)^*) \to Store' \to Store' \\
Acc & : & ((PROC \times \mathcal{P}(LPROC)^*) \to Store' \to Store' \times \\
 & & \quad \times \mathcal{P}(LPROC) \times \mathcal{P}(LPROC)^*) \to Store' \to Store' \\
Union & : & \widehat{Venv'} \to Store' \to Store' \\
UnionN & : & (VAR \times \mathcal{P}(LPROC))^* \to Store' \to Store'
\end{array}
$$

Figure 3.1: State passing 0-CFA: functionalities

Store is the set of possible states. For finiteness reasons, we need to restrict it. Fixing $VAR = \{x_1, ..., x_n\}$ and $CALL = \{c_1, ..., c_m\}$, we define the functions:

$$
\begin{array}{rl}
Size : \widehat{Venv} \to \mathbf{N} & Size(\widehat{\rho}) = |\widehat{\rho}(x_1)| + |\widehat{\rho}(x_2)| + ... + |\widehat{\rho}(x_n)| \\
MaxT : Times \to \mathbf{N} & MaxT(\widehat{\tau}) = max\,(\widehat{\tau}(c_1), \widehat{\tau}(c_2), ..., \widehat{\tau}(c_m))
\end{array}
$$

We define $Store' \subset Store$ as

$$Store' = \left\{ \left(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t\right) \in Store \mid t \leq Size(\widehat{\rho}) \wedge MaxT(\widehat{\tau}) \leq t \right\}$$

From the definition, *Size* is a bounded function. As a consequence *Store'* is a finite set.

Informally, the restriction comes from the fact that the time counter cannot be increased indefinitely, because the variable environment cannot be increased indefinitely. It also accounts for the fact that the memorized time stamp of each call site is less than or equal to the current time.

The definition of the new analysis can be found in Figure 3.2. Again, here we consider the domains of definitions as complete lattices. The ordering on states from *Store'*, is considered as the one induced when considering time values ordered as natural numbers. The recursive equations give values to $\widehat{\mathcal{F}}'$ only on a subset $RFPAR' \subset (PROC \times \mathcal{P}(LPROC)^*)$ defined as

$$(l, \langle A_1, ..., A_n \rangle) \in RFPAR' \iff arity(l) = n$$

It is easy to prove that we are only calling $\widehat{\mathcal{F}}$ on values inside $RFPAR'$. Unlike the definition in Chapter 2, where we defined the analysis only on the restriction, here we consider $\widehat{\mathcal{F}}'$ to be defined on the entire domain, and by definition it is returning $\emptyset$ on values outside the restriction.

We prove the existence of the functions defined in Figure 3.2. First, $\widehat{\mathcal{A}}'$ is obviously well-defined. The function *Acc* can raise some questions, due to the fact that different choices of orders on the set of functions received might yield different values. However, this can be easily fixed by defining an order on the elements in *PROC* and defining *Acc* to always pick that ordering. We do not go more into details, it is not a relevant part of our argument.

Union is well-defined by the following lemma.

Lemma 3.2.1. *For all* $\left(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t\right) \in Store'$ *and for all* $\widehat{\rho}_1 \in \widehat{Venv}$,

$$\mathsf{Union}(\widehat{\rho}_1)\left(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t\right) \in Store'$$

Proof. Two cases occur. If $\widehat{\rho}_1 \sqsubseteq \widehat{\rho}$ then the above obviously holds. Otherwise, if $\widehat{\rho}_1 \not\sqsubseteq \widehat{\rho}$ then we can easily see that

$$t \leq Size(\widehat{\rho}) < Size(\widehat{\rho} \sqcup \widehat{\rho}_1)$$

$$
\begin{array}{rcl}
\widehat{\mathcal{A}}'[\![k]\!]\big(\widehat{\rho},\widehat{\delta},\widehat{\tau},t\big) & = & \emptyset \\
\widehat{\mathcal{A}}'[\![p{:}\texttt{if}]\!]\big(\widehat{\rho},\widehat{\delta},\widehat{\tau},t\big) & = & \{p\} \\
\widehat{\mathcal{A}}'[\![l{:}\lambda]\!]\big(\widehat{\rho},\widehat{\delta},\widehat{\tau},t\big) & = & \{l\} \\
\widehat{\mathcal{A}}'[\![x]\!]\big(\widehat{\rho},\widehat{\delta},\widehat{\tau},t\big) & = & \widehat{\rho}(x)
\end{array}
$$

$$
\begin{array}{rcl}
\widehat{\mathcal{C}}'\ [\![c{:}\texttt{(}f\ a_1\ ...\ a_n\texttt{)}]\!]\ \big(\widehat{\rho},\widehat{\delta},\widehat{\tau},t\big) & = & \textbf{if}\ \widehat{\tau}(c) < t\ \textbf{then} \\
 & & \quad Acc\Big(\widehat{\mathcal{F}}', F, \langle A_1, ..., A_n\rangle\Big)\ \big(\widehat{\rho},\widehat{\delta}_1,\widehat{\tau}_1,t\big) \\
 & & \quad\quad \text{where}\ F = \widehat{\mathcal{A}}'[\![f]\!]\big(\widehat{\rho},\widehat{\delta},\widehat{\tau},t\big) \\
 & & \quad\quad\quad A_i = \widehat{\mathcal{A}}'[\![a_i]\!]\big(\widehat{\rho},\widehat{\delta},\widehat{\tau},t\big) \\
 & & \quad\quad\quad \widehat{\delta}_1 = \widehat{\delta} \sqcup [c \mapsto F], \widehat{\tau}_1 = \widehat{\tau}[c \mapsto t] \\
 & & \textbf{else}\ \big(\widehat{\rho},\widehat{\delta},\widehat{\tau},t\big)
\end{array}
$$

$$
\begin{array}{l}
\widehat{\mathcal{C}}'\ [\![c{:}\texttt{(letrec ((}x_1\ d_1\texttt{)}...\texttt{(}x_n\ d_n\texttt{))}\ e\texttt{)}]\!]\ \big(\widehat{\rho},\widehat{\delta},\widehat{\tau},t\big) \ = \ \widehat{\mathcal{C}}'\ [\![e]\!]\ \big(\widehat{\rho}_1,\widehat{\delta}_1,\widehat{\tau}_1,t_1\big) \\
\quad \text{where}\ \big(\widehat{\rho}_1,\widehat{\delta}_1,\widehat{\tau}_1,t_1\big) = UnionN\langle (x_1,A_1), ..., (x_n,A_n)\rangle\big(\widehat{\rho},\widehat{\delta},\widehat{\tau},t\big) \\
\quad\quad A_i = \widehat{\mathcal{A}}'[\![d_i]\!]\big(\widehat{\rho},\widehat{\delta},\widehat{\tau},t\big)
\end{array}
$$

$$
\begin{array}{rcl}
\widehat{\mathcal{F}}'\ [\![l{:}\texttt{(}\lambda\ \texttt{(}x_1\ ...\ x_n\texttt{)}\ e\texttt{)}]\!]\ \langle A_1,...,A_n\rangle\ \big(\widehat{\rho},\widehat{\delta},\widehat{\tau},t\big) & = & \widehat{\mathcal{C}}'\ [\![e]\!]\ \big(\widehat{\rho}_1,\widehat{\delta}_1,\widehat{\tau}_1,t_1\big) \\
\multicolumn{3}{r}{\text{where}\ \big(\widehat{\rho}_1,\widehat{\delta}_1,\widehat{\tau}_1,t_1\big) = UnionN\langle (x_1,A_1), ..., (x_n,A_n)\rangle\big(\widehat{\rho},\widehat{\delta},\widehat{\tau},t\big)} \\
\widehat{\mathcal{F}}'\ [\![p{:}\texttt{if}]\!]\ \langle X, K_1, K_2\rangle\ \big(\widehat{\rho},\widehat{\delta},\widehat{\tau},t\big) & = & Acc(\widehat{\mathcal{F}}', K_1 \cup K_2, \langle\rangle)\big(\widehat{\rho},\widehat{\delta}_1,\widehat{\tau},t\big) \\
 & & \text{where}\ \widehat{\delta}_1 = \widehat{\delta} \sqcup [\texttt{if}_p^t \mapsto K_1, \texttt{if}_p^f \mapsto K_2] \\
\widehat{\mathcal{F}}'\ [\![p{:}\texttt{+}]\!]\ \langle X, Y, K\rangle\ \big(\widehat{\rho},\widehat{\delta},\widehat{\tau},t\big) & = & Acc(\widehat{\mathcal{F}}', K, \langle\emptyset\rangle)\big(\widehat{\rho},\widehat{\delta}_1,\widehat{\tau},t\big) \\
 & & \quad\quad \text{where}\ \widehat{\delta}_1 = \widehat{\delta} \sqcup [\texttt{+}_p \mapsto K] \\
\widehat{\mathcal{F}}'\ Stop\ \langle A_1,...,A_n\rangle\ \big(\widehat{\rho},\widehat{\delta},\widehat{\tau},t\big) & = & \big(\widehat{\rho},\widehat{\delta},\widehat{\tau},t\big)
\end{array}
$$

$$
\begin{array}{rll}
Acc(\Phi, S, \langle A_1,...,A_m\rangle)\sigma^0 = & \textbf{let}\ \sigma^1 = & \textbf{if}\ \text{arity}(f_1) = m\ \textbf{then} \\
 & & \quad \Phi\ \ulcorner f_1 \urcorner\ \langle A_1,...,A_m\rangle\ \sigma^0 \\
 & & \textbf{else}\ \sigma^0 \\
 & & ... \\
 & \quad\quad \sigma^n = & \textbf{if}\ \text{arity}(f_n) = m\ \textbf{then} \\
 & & \quad \Phi\ \ulcorner f_n \urcorner\ \langle A_1,...,A_m\rangle\ \sigma^{n-1} \\
 & & \textbf{else}\ \sigma^{n-1} \\
 & \textbf{in}\ \sigma^n &
\end{array}
$$

where $f_1, ..., f_n$ is a fixed arbitrary enumeration of the finite set S.

$$
\begin{array}{c}
Union(\widehat{\rho}_1)\big(\widehat{\rho},\widehat{\delta},\widehat{\tau},t\big) = \textbf{if}\ \widehat{\rho}_1 \sqsubseteq \widehat{\rho}\ \textbf{then}\ \big(\widehat{\rho},\widehat{\delta},\widehat{\tau},t\big)\ \textbf{else}\ (\widehat{\rho} \sqcup \widehat{\rho}_1, \widehat{\delta}, \widehat{\tau}, t+1) \\
UnionN\langle (x_1,A_1), ..., (x_n,A_n)\rangle = Union([x_1 \mapsto A_1]) \circ ... \circ Union([x_n \mapsto A_n])
\end{array}
$$

Figure 3.2: State-passing 0-CFA: the equations

from which we deduce that $t+1 \leq Size(\widehat{\rho} \sqcup \widehat{\rho}_1)$, obtaining that

$$Union(\widehat{\rho}_1)\big(\widehat{\rho},\widehat{\delta},\widehat{\tau},t\big) \in Store'$$

□

We can use Lemma 3.2.1 to prove a similar property for $UnionN$: for all $x_1, ..., x_n \in VAR$, $A_1, ..., A_n \in \mathcal{P}(LPROC)$ and $\sigma \in Store'$ the following holds

$$UnionN(\langle (x_1,A_1), ..., (x_n,A_n)\rangle)\sigma \in Store'$$

which tells us that $UnionN$ is also well-defined.

To define $\widehat{\mathcal{C}}', \widehat{\mathcal{F}}'$, Let

$$
\begin{array}{rl}
CSPACE' = & CALL \to Store' \to Store' \\
FSPACE' = & (PROC \times \mathcal{P}(LPROC)^*) \to Store' \to Store'
\end{array}
$$

We consider again an operator on complete lattices $H' : (\mathit{CSPACE}' \times \mathit{FSPACE}') \to (\mathit{CSPACE}' \times \mathit{FSPACE}')$ defined in the same way as H in Chapter 2: for a pair $\left(\widehat{\mathcal{C}}', \widehat{\mathcal{F}}'\right)$ we obtain $H'\left(\widehat{\mathcal{C}}', \widehat{\mathcal{F}}'\right)$ from the definitions in Figure 3.2, where on the right-hand side we use $\widehat{\mathcal{C}}'$ and $\widehat{\mathcal{F}}'$.

Lemma 3.2.2. *H' is well-defined.*

Proof. We need to prove that for an arbitrary pair $\left(\widehat{\mathcal{C}}', \widehat{\mathcal{F}}'\right) \in (\mathit{CSPACE}' \times \mathit{FSPACE}')$, also $H'\left(\widehat{\mathcal{C}}', \widehat{\mathcal{F}}'\right) \in (\mathit{CSPACE}' \times \mathit{FSPACE}')$. Let $H'\left(\widehat{\mathcal{C}}', \widehat{\mathcal{F}}'\right) = \left(\widehat{\mathcal{C}}'_1, \widehat{\mathcal{F}}'_1\right)$.

Take an arbitrary $C \in \mathit{CALL}$ and $\left(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t\right) \in \mathit{Store}'$. We show that $\widehat{\mathcal{C}}'_1\ C\ \left(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t\right) \in \mathit{Store}'$. Two cases occur:

- C is an application site. Assume that $\widehat{\tau}(C) < t$ (the other case is trivial). Then:

$$\widehat{\mathcal{C}}'_1\ C\ \left(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t\right) = \mathit{Acc}(\widehat{\mathcal{F}}', F, A_1, ..., A_n)\left(\widehat{\rho}, \widehat{\delta}_1, \widehat{\tau}_1, t\right)$$

  for $\left(\widehat{\rho}, \widehat{\delta}_1, \widehat{\tau}_1, t\right) \in \mathit{Store}'$. We can use the hypothesis on $\widehat{\mathcal{F}}'$ to show that the above $\in \mathit{Store}'$.

- C is a `letrec` construct. We get:

$$\widehat{\mathcal{C}}'_1\ [\![c\texttt{:(letrec ((}x_1\ d_1\texttt{)...(}x_n\ d_n\texttt{))}\ e\texttt{)}]\!]\ \left(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t\right) = \widehat{\mathcal{C}}'\ [\![e]\!]\ \left(\widehat{\rho}_1, \widehat{\delta}_1, \widehat{\tau}_1, t_1\right)$$

  which is again is a member of Store', from the hypothesis and Lemma 3.2.1

Similar arguments can be used for $\widehat{\mathcal{F}}'_1$. □

Lemma 3.2.3. *H' is monotone.*

It follows easily from the definition.

The domain of H' is finite. From its monotonicity, its least fixed point exists and can be obtained as the limit of the chain of approximations:

$$\begin{array}{rcl}
\left(\widehat{\mathcal{C}}'_i, \widehat{\mathcal{F}}'_i\right)_{i \in \mathbf{N}} & \in & (\mathit{CSPACE}' \times \mathit{FSPACE}') \\
\left(\widehat{\mathcal{C}}'_0, \widehat{\mathcal{F}}'_0\right) & = & (\lambda c.\lambda\sigma.\bot, \lambda(f, L).\lambda\sigma.\bot) \\
\left(\widehat{\mathcal{C}}'_{i+1}, \widehat{\mathcal{F}}'_{i+1}\right) & = & H'\left(\widehat{\mathcal{C}}'_i, \widehat{\mathcal{F}}'_i\right)
\end{array}$$

Considering $\left(\widehat{\mathcal{C}}', \widehat{\mathcal{F}}'\right)$ to be the limit of this chain, we define that the analysis for a program *Prg* is given by:

$$\widehat{\mathcal{F}}'\ \mathit{Prg}\ \langle\{\mathit{Stop}\}\rangle\ \ (\emptyset, \emptyset, 0_{\mathit{CALL}}, 0)$$

where $0_{\mathit{CALL}} : \mathit{CALL} \to \mathbf{N}$ defined such that $\forall c \in \mathit{CALL}, 0_{\mathit{CALL}}(c) = 0$.

3.3 Relating the Analyses

Let us relate now the analysis defined in Figure 3.2 with Shivers's 0-CFA. The result of the new 0-CFA semantics is to be less precise but as a direct consequence it is still a safe approximation.

The results computed by the two analyses are not directly comparable. However, we can naturally define an ordering relation between them, converting labels into terms in the source program:

Definition 3.3.1. *We define the binary relation $\sqsubseteq^C \subset (\widehat{\mathit{CCache}} \times \mathit{Store}')$ such that $\widehat{\delta} \sqsubseteq^C \left(\widehat{\rho}', \widehat{\delta}', \widehat{\tau}', t'\right)$ iff:*

$$\forall l \in \mathit{LAB}.\widehat{\delta}(l) \subseteq \ulcorner\widehat{\delta}'(l)\urcorner$$

A similar ordering can be defined among the variable environments:

Definition 3.3.2. *We define the relation* $\sqsubseteq^V \subset (\widehat{Venv} \times \widehat{Venv'})$ *and we write* $\widehat{\rho} \sqsubseteq^V \widehat{\rho}'$ *if:*

$$\forall x \in VAR.\widehat{\rho}(x) \subseteq \ulcorner \widehat{\rho}'(x) \urcorner$$

The relationship between the two analyses is given by the following

Theorem 3.3.3.

$$\left(\widehat{\mathcal{F}}\ Prg\ \langle\{Stop\}\rangle\ \emptyset\right) \sqsubseteq^C \left(\widehat{\mathcal{F}}'\ Prg\ \langle\{Stop\}\rangle\ (\emptyset, \emptyset, 0_{CALL}, 0)\right)$$

The proof is done by relating both the analysis with an intermediate version. Similar with Shivers's informal argument, we first show that introducing time-stamps in the original equations is not altering the result of the analysis. Then we show that using a global environment, together with the time-stamps technique computes a safe approximation of the analysis.

We say that $(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t) \in Store'$ is valid if $\forall x \in VAR.\ulcorner\widehat{\rho}(x)\urcorner \in \mathcal{P}(LAM)$.

Lemma 3.3.4. *For all* $f \in PROC$, $A_1, ..., A_n \in LPROC$ *such that* $\ulcorner A_i \urcorner \in \mathcal{P}(LAM)$ *and* $(f, A_1, ..., A_n) \in RFPAR'$, *and* σ *valid,*

$$\left(\widehat{\mathcal{F}}'\ f\ \langle A_1, ..., A_n\rangle\ \sigma\right) = \sigma'$$

such that σ' *valid and* $\sigma \sqsubseteq \sigma'$.

We define first a well-founded relation on elements of $Store'$.

Definition 3.3.5. *For* $(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t), (\widehat{\rho}_1, \widehat{\delta}_1, \widehat{\tau}_1, t_1) \in Store'$ *we define that the relation* $(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t) \prec (\widehat{\rho}_1, \widehat{\delta}_1, \widehat{\tau}_1, t_1)$ *holds if an only if*

$$(\widehat{\rho}_1 \sqsubset \widehat{\rho} \wedge t_1 < t) \vee ((\widehat{\rho}, t) = (\widehat{\rho}_1, t_1) \wedge \exists c \in CALL.\widehat{\tau}_1(c) < \widehat{\tau}(c) = t)$$

It is immediate to prove that $\prec$ is a well-founded relation on $Store'$.

Proof of Lemma 3.3.4. Defining the statement of the lemma as a predicate Q on σ, we prove that Q holds for all σ.

We use well-founded induction. We prove that

$$\forall \sigma \in Store'.(\forall \sigma' \prec \sigma.Q(\sigma')) \Rightarrow Q(\sigma)$$

Consider an arbitrary valid σ. Assume $(\forall \sigma' \prec \sigma.Q(\sigma'))$. We prove $Q(\sigma)$. Take first $f \in LAM$ and $A_1, ..., A_n \in LPROC$ such that $\ulcorner A_i \urcorner \in \mathcal{P}(LAM)$ and $(f, \langle A_1, ..., A_n\rangle) \in RFPAR'$. Assuming $f \equiv l$`:(`λ `(`x_1 ... x_n`)` e`)`, we can see from the fixed point property that:

$$\widehat{\mathcal{F}}'\ f\ \langle A_1, ..., A_n\rangle\ \sigma = \widehat{\mathcal{C}}'\ [\![e]\!]\ \sigma_1$$

where $\sigma_1 = UnionN\langle(x_1, A_1), ..., (x_n, A_n)\rangle\sigma$. It is easy to see that σ_1 is also valid and that $\sigma_1 \prec \sigma$ or $\sigma_1 = \sigma$. We do now a case analysis on e.

Assume e is of the form c`:(`f a_1 ... a_n`)`. Then:

- If $\widehat{\tau}_1(e) = t_1$ then $Q(\sigma)$ holds trivially.
- If $\widehat{\tau}_1(e) < t_1$ then we obtain

  $$Acc\left(\widehat{\mathcal{F}}', F, \langle B_1, ..., B_n\rangle\right)\sigma_2$$

  where we can prove that $\ulcorner B_i \urcorner \in \mathcal{P}(LAM)$ and that $\sigma_2 \prec \sigma_1$ and σ_2 valid. By directly using the hypothesis we easily prove that $Q(\sigma)$ holds.

$$\begin{array}{lcl}
\widehat{\mathcal{C}}^* \ [\![c\text{:}(f\ a_1\ ...\ a_n)]\!]\ \big(\widehat{\rho}, \widehat{\tau}, t\big) & = & \textbf{if } \widehat{\tau}(c) < t \textbf{ then} \\
& & \bigsqcup_{f \in F} \Big(\widehat{\mathcal{F}}^*\ f\ \langle A_1, ..., A_n\rangle\ \big(\widehat{\rho}, \widehat{\tau}_1, t\big)\Big) \sqcup [c \mapsto F] \\
& & \quad \text{where } F = \widehat{\mathcal{A}}^*[\![f]\!]\big(\widehat{\rho}, \widehat{\tau}, t\big), A_i = \widehat{\mathcal{A}}^*[\![a_i]\!]\big(\widehat{\rho}, \widehat{\tau}, t\big) \\
& & \qquad\quad \widehat{\tau}_1 = \widehat{\tau}[c \mapsto t] \\
& & \textbf{else } \emptyset
\end{array}$$

$$\widehat{\mathcal{F}}^*\ [\![l\text{:}(\lambda\ (x_1\ ...\ x_n)\ e)]\!]\ \langle A_1, ..., A_n\rangle\ \big(\widehat{\rho}, \widehat{\tau}, t\big) \quad = \quad \widehat{\mathcal{C}}^*\ [\![e]\!]\ \big(\widehat{\rho}_1, \widehat{\tau}_1, t_1\big)$$
$$\text{where } \big(\widehat{\rho}_1, \_, \widehat{\tau}_1, t_1\big) = \mathit{UnionN}\langle(x_1, A_1), ..., (x_n, A_n)\rangle\big(\widehat{\rho}_1, \_, \widehat{\tau}_1, t_1\big)$$

Figure 3.3: Introducing time-stamps

In the other case of e, we need to construct a recursive judgment on the structure of e, which ends with a prove as above.

We prove now that the result holds also for $f \in \mathit{PRIM}$. In all possible cases of f, the above expands to an expression dependent on $\widehat{\mathcal{F}}'$ applied to $f \in \mathit{LAM}$ (n.b.: and here is where the notion of valid pays off), for which we have proven above that the result holds.

The case $f = \mathit{Stop}$ is trivial. Lemma 3.3.4 thus holds. □

We introduce an intermediate set of control-flow equations, given in the form of a pair $\big(\widehat{\mathcal{C}}^*, \widehat{\mathcal{F}}^*\big)$, which, compared with Shivers's original, operate with time-stamps but do not sequentialize the computation, as the one in Figure 3.2.

In Figure 3.3 we present the essential modifications. For simplicity, we analyze only the application and lambda cases, as the results can be easily extended to the other cases. The domains and functionalities should be obvious. With a similar argument as before, the existence of the functions is established.

Lemma 3.3.6. *For all $\big(\widehat{\rho}, \widehat{\tau}, t\big)$ which satisfies the constraints of Store' and $c \in \mathit{CALL}$, if $S = \{c \in \mathit{CALL} | \tau(c) = t\}$ then*

$$\widehat{\mathcal{C}}^*\ c\ \big(\widehat{\rho}, \widehat{\tau}, t\big) \sqcup \left(\bigsqcup_{c' \in S} \widehat{\mathcal{C}}\ c'\ \widehat{\rho}\right) = \widehat{\mathcal{C}}\ c\ \widehat{\rho} \sqcup \left(\bigsqcup_{c' \in S} \widehat{\mathcal{C}}\ c'\ \widehat{\rho}\right)$$

Proof. We reuse the well-founded relation from Definition 3.3.5. Since it is independent on $\widehat{\delta}$, the relation $\prec$ can be projected to tuples $\big(\widehat{\rho}, \widehat{\tau}, t\big)$ preserving the well-foundedness property.

We prove the lemma by well-founded induction. Taking Q to be the statement of the lemma as a predicate on $\sigma = \big(\widehat{\rho}, \widehat{\tau}, t\big)$, we prove that

$$\forall \sigma \in \mathit{Store}'.(\forall \sigma' \prec \sigma. Q(\sigma')) \Rightarrow Q(\sigma)$$

Again, take an arbitrary $\sigma = \big(\widehat{\rho}, \widehat{\tau}, t\big)$ and assume $(\forall \sigma' \prec \sigma. Q(\sigma'))$. We want to prove that for arbitrary $c \in \mathit{CALL}$,

$$\widehat{\mathcal{C}}^*\ c\ \big(\widehat{\rho}, \widehat{\tau}, t\big) \sqcup \left(\bigsqcup_{c' \in S} \widehat{\mathcal{C}}\ c'\ \widehat{\rho}\right) = \widehat{\mathcal{C}}\ c\ \widehat{\rho} \sqcup \left(\bigsqcup_{c' \in S} \widehat{\mathcal{C}}\ c'\ \widehat{\rho}\right)$$

If $\widehat{\tau}(c) = t$, then $c \in S$ and the above holds trivially. If $\widehat{\tau}(c) < t$, the above is equivalent (for appropriate F) with

$$\bigsqcup_{f \in F} \Big(\widehat{\mathcal{F}}^*\ f\ \langle A_1, ..., A_n\rangle\ \big(\widehat{\rho}, \widehat{\tau}_1, t\big)\Big) \sqcup [c \mapsto F] \sqcup \left(\bigsqcup_{c' \in S} \widehat{\mathcal{C}}\ c'\ \widehat{\rho}\right) =$$
$$= \bigsqcup_{f \in F} \Big(\widehat{\mathcal{F}}\ f\ \langle A_1, ..., A_n\rangle\ \widehat{\rho}\Big) \sqcup [c \mapsto F] \sqcup \left(\bigsqcup_{c' \in S} \widehat{\mathcal{C}}\ c'\ \widehat{\rho}\right)$$

where $\left(\widehat{\rho}, \widehat{\tau}_1, t\right) \prec \sigma$. The above is again equivalent (for an appropriate C) with

$$\begin{aligned}
&\bigsqcup_{c'' \in C} \left(\widehat{\mathcal{C}}^* \; c'' \; \left(\widehat{\rho}_{c''}, \widehat{\tau}_{c''}, t_{c''}\right)\right) \sqcup [c \mapsto F] \sqcup \left(\bigsqcup_{c' \in S} \widehat{\mathcal{C}} \; c' \; \widehat{\rho}\right) = \\
&\qquad = \bigsqcup_{c'' \in C} \left(\widehat{\mathcal{C}} \; c'' \; \widehat{\rho}_{c''}\right) \sqcup [c \mapsto F] \sqcup \left(\bigsqcup_{c' \in S} \widehat{\mathcal{C}} \; c' \; \widehat{\rho}\right)
\end{aligned}$$

where $\forall c'' \in C$, $\left(\widehat{\rho}_{c''}, \widehat{\tau}_{c''}, t_{c''}\right) \prec \sigma$ and C is obtained from F. Here we can now use the well-founded induction hypothesis to prove the equality, doing a case analysis on $t_{c''}$ compared to t. We do not go in to details. $\square$

To relate the "intermediate" analysis with the state-passing one, we prove

Lemma 3.3.7. *For all $\left(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t\right) \in Store'$, $(f, \langle A_1, ..., A_n\rangle) \in RFPAR'$ we have*

$$\widehat{\mathcal{F}}^* \; f \; \langle A_1, ..., A_n\rangle \; \left(\widehat{\rho}, \widehat{\tau}, t\right) \sqsubseteq^C \widehat{\mathcal{F}}' \; f \; \langle A_1, ..., A_n\rangle \; \left(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t\right)$$

Proof. We prove the above by the same technique of well-founded induction over the well-founded order relation introduced in Definition 3.3.5.

Taking an arbitrary $\sigma = \left(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t\right)$, assuming that the result holds for $\sigma' \prec \sigma$, we are left to prove that:

$$\widehat{\mathcal{C}}^* \; c \; \left(\widehat{\rho}_a, \widehat{\tau}_a, t_a\right) \sqsubseteq^C \widehat{\mathcal{C}}' \; c \; \left(\widehat{\rho}_a, \widehat{\delta}_a, \widehat{\tau}_a, t_a\right)$$

The case $\widehat{\tau}(c) = t$ is trivial. Assuming the opposite, the above leads to:

$$\bigsqcup_{f \in F} \widehat{\mathcal{F}}^* \; f \; \langle B_1, ..., B_n\rangle \; \left(\widehat{\rho}_0, \widehat{\tau}_0, t_0\right) \sqsubseteq^C Acc\left(\widehat{\mathcal{F}}', F, \langle B_1, ..., B_n\rangle\right) \left(\widehat{\rho}_0, \widehat{\delta}_0, \widehat{\tau}_0, t_0\right)$$

where $\left(\widehat{\rho}_0, \widehat{\delta}_0, \widehat{\tau}_0, t_0\right) \prec \left(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t\right)$.

Assuming $F = \{f_1, ..., f_m\}$, we define:

$$\begin{array}{rcl}
\left(\widehat{\rho}_1, \widehat{\delta}_1, \widehat{\tau}_1, t_1\right) & = & \widehat{\mathcal{F}}' \; f_1 \; \langle B_1, ..., B_n\rangle \; \left(\widehat{\rho}_0, \widehat{\delta}_0, \widehat{\tau}_0, t_0\right) \\
\left(\widehat{\rho}_2, \widehat{\delta}_2, \widehat{\tau}_2, t_2\right) & = & \widehat{\mathcal{F}}' \; f_1 \; \langle B_1, ..., B_n\rangle \; \left(\widehat{\rho}_1, \widehat{\delta}_1, \widehat{\tau}_1, t_1\right) \\
 & \vdots & \\
\left(\widehat{\rho}_m, \widehat{\delta}_m, \widehat{\tau}_m, t_m\right) & = & \widehat{\mathcal{F}}' \; f_m \; \langle B_1, ..., B_n\rangle \; \left(\widehat{\rho}_{m-1}, \widehat{\delta}_{m-1}, \widehat{\tau}_{m-1}, t_{m-1}\right)
\end{array}$$

By the induction hypothesis we have that for all $0 < i < m$

$$\widehat{\mathcal{F}}^* \; f_i \; \langle B_1, ..., B_n\rangle \; \left(\widehat{\rho}_0, \widehat{\tau}_0, t_0\right) \sqsubseteq^C \widehat{\mathcal{F}}' \; f_i \; \langle B_1, ..., B_n\rangle \; \left(\widehat{\rho}_0, \widehat{\delta}_0, \widehat{\tau}_0, t_0\right)$$

We just need to show that

$$\widehat{\mathcal{F}}' \; f_i \; \langle B_1, ..., B_n\rangle \; \left(\widehat{\rho}_0, \widehat{\delta}_0, \widehat{\tau}_0, t_0\right) \sqsubseteq \widehat{\mathcal{F}}' \; f_i \; \langle B_1, ..., B_n\rangle \; \left(\widehat{\rho}_i, \widehat{\delta}_i, \widehat{\tau}_i, t_i\right)$$

for all $0 < i < m$.

The above can be proven by induction on i. For space reasons we do not present the proof here. It is based on the fact that in the case of $\widehat{\mathcal{F}}'$, we can identify all the intermediate states in the computation between $\left(\widehat{\rho}_0, \widehat{\delta}_0, \widehat{\tau}_0, t_0\right)$ and $\left(\widehat{\rho}_1, \widehat{\delta}_1, \widehat{\tau}_1, t_1\right)$, for instance, as a finite ascending chain, and we can pinpoint all cases where the environment or the call cache are increased. $\square$

As a direct consequence of the above lemma and of the result obtained from Lemma 3.3.6, we obtain

Proof of Theorem 3.3.3. A direct consequence of Lemma 3.3.6 is that

$$\left(\widehat{\mathcal{F}}\ \mathit{Prg}\ \langle\{\mathit{Stop}\}\rangle\ \emptyset\right) = \left(\widehat{\mathcal{F}}^{*}\ \mathit{Prg}\ \langle\{\mathit{Stop}\}\rangle\ \ (\emptyset, 0_{CALL}, 0)\right)$$

which tells us that the results Shivers's analysis is equal with the one given by the intermediate one. From Lemma 3.3.7 we obtain that the state-passing analysis is safe with respect to the intermediate one:

$$\left(\widehat{\mathcal{F}}^{*}\ \mathit{Prg}\ \langle\{\mathit{Stop}\}\rangle\ \ (\emptyset, 0_{CALL}, 0)\right) \sqsubseteq^{C} \widehat{\mathcal{F}}'\ \mathit{Prg}\ \langle\{\mathit{Stop}\}\rangle\ \ (\emptyset, \emptyset, 0_{CALL}, 0)$$

The two observations yield the statement of the theorem. □

3.4 Summary and Conclusion

In this chapter we have modified Shivers's analysis to a formulation close to the actual implementation. We have proven the correctness of our analysis by proving that the result obtained contains the result obtained by Shivers's original 0-CFA.

The purpose of defining the new analysis is to have a specification sufficiently close to the intended implementation. It enables us to code the analysis it in a programming language directly and efficiently, and still prove the correctness of implementation. This idea is developed in the following chapters. The design of a state-passing like form of the equations is closely linked to the design of a small subset of ML which will serve as our implementation language.

In the next chapter, we are formally describing the language, and we present a program computing the modified analysis introduced here. Using the modified version of the analysis and the formal semantics of the language we are able to prove correctness of our implementation.

Chapter 4

Our Implementation Language

4.1 Introduction

The 0-CFA equations from Figure 3.2 in the previous chapter have been designed such that they can be implemented in a general-purpose programming language. In this chapter we formalize a subset of the implementation language (ML in this case) which is sufficient for programming the analysis and its specialized version.

In fact, we consider our implementation language as a framework for reasoning about the implementation of the analysis and its specialized versions, as well as about valid program transformations. This approach unifies our treatment of implementation and program specialization, in the same time providing a correctness proof of the actual programs used for benchmarking.

4.2 Syntax

The language, called IMP, is a typed language of procedures and commands. It allows top-level declarations of recursive procedures built from successions of commands operating on a global store. Declared procedures take only arguments of ground types, but higher-order values are allowed in computation.

For simplicity, we consider that the state is indirectly modified through built in commands. Case instructions on syntax of the CPS terms are also instantiations of the general ML case construct over a user-defined datatype. Constants are added for all the mathematical functions of which we make use in Figure 3.2.

IMP is monomorphically typed. Let us start by defining a language of types. The atomic types b, base types β and types τ are defined by the grammar in Figure 4.1.

The syntax of the language is given in Figure 4.2. Each constant in the language is defined with an associated type. The list of predefined constants and their associated types is given in Figure 4.3.

We consider valid programs to be the ones that are in the typing relation. Several typing relations are introduced over the different syntactic categories as follows:

$$\begin{array}{lll} P : \mathsf{unit} & & \Gamma \vdash_E E : \tau \quad , E \in \mathit{Exp} \\ \Gamma \vdash_C C : \mathsf{unit} \quad , C \in \mathit{Com} & & \Gamma \vdash_V V : \tau \quad , V \in \mathit{Val} \end{array}$$

The typing relations are derived from the typing rules given in Figure 4.4. For a constant $c : \tau$, the relation $\Gamma \vdash_V c : \tau$ holds by definition for all Γ.

In addition, given the source program *Prg* to analyze, the language IMP has constants of appropriate type for each sub-term of *Prg*, which are intended to denote the respective terms. For instance, given the term l:$(\lambda\ (id_1\ ...\ id_n)\ e) \in \mathit{PROC}$ we have constants $\mathtt{id}_i : \mathsf{VAR}$, $\mathtt{l} : \mathsf{LAB}$, $\mathtt{e} : \mathsf{CALL}$. Similarly, for a term c:$(f\ a_1\ ...\ a_n) \in \mathsf{CALL}$, we have constants $\mathtt{a}_i, \mathtt{f} : \mathsf{ARGFUN}$, $\mathtt{c} : \mathsf{LAB}$, and so on. We consider $\mathtt{Prg} : \mathsf{PROC}$ to be a constant denoting *Prg*.

$$
\begin{array}{lcl}
b & ::= & \textsf{nat} \mid \textsf{bool} \mid \textsf{PROC} \mid \textsf{LPROC} \mid \\
 & & \textsf{LPROC set} \mid \textsf{ARGFUN} \mid \textsf{VAR} \mid \textsf{CALL} \mid \textsf{LAB} \\
\beta & ::= & b \mid b\ \textsf{list} \\
\tau & ::= & \textsf{unit} \mid \beta \mid (\tau_1 \times ... \times \tau_i)_{i\in\mathbf{N}} \mid \tau_1 \to \tau_2
\end{array}
$$

Figure 4.1: Types of IMP

$$
\begin{array}{rl}
P ::= & \texttt{letrec } D^* \texttt{ in } C \\
D \in \mathit{Def} ::= & \texttt{fun } f\texttt{=}(x_1^{\beta_1}, ..., x_n^{\beta_n}) \texttt{ = } C \\
 & \\
C \in \mathit{Com} ::= & V\texttt{(}E_1, ..., E_n\texttt{)} \mid \texttt{skip} \mid \\
 & \texttt{(}C_1\texttt{;}...\texttt{;}C_n\texttt{)} \mid \\
 & \texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2 \mid \\
 & \texttt{caseCall } V \texttt{ of} \\
 & \quad l_1\texttt{:APP(}e, es\texttt{)=> } C_1 \\
 & \quad \texttt{|}l_2\texttt{:REC(}ids, es, e\texttt{)=> } C_2 \mid \\
 & \texttt{caseProc } V \texttt{ of} \\
 & \quad l_1\texttt{:LAM(}ids, e\texttt{)=> } C_1 \\
 & \quad \texttt{|}l_2\texttt{:BIF => } C_2 \\
 & \quad \texttt{|}l_3\texttt{:BPLUS => } C_3 \\
 & \quad \texttt{| Stop => } C_4
\end{array}
\qquad
\begin{array}{rl}
E \in \mathit{Exp} ::= & V\texttt{(}V_1\texttt{, . . . ,}V_n\texttt{)} \mid \\
 & \texttt{[}E_1\texttt{, . . . ,}E_n\texttt{]} \mid \\
 & V \\
V \in \mathit{Val} ::= & c \mid x \\
c \in \mathit{Const} & \\
x, f, ids, es, e \in \mathit{Var} &
\end{array}
$$

Figure 4.2: Syntax of IMP programs

We are not very precise about the conversion between terms in *Prg* and their respective constants in the language IMP. When using a term of *Prg* inside a program in IMP, we consider it as the associated constant.

We do not always specify the types in the function declarations in IMP programs, as required by the syntax. In all of the cases, they can be deduced from the context.

4.3 Semantics

We give a denotational semantics to valid IMP programs as a standard store-passing semantics. The programs operate over a global store, which is by definition an element of *Store*′. We start by giving a interpretation of types, assigning a cpo to each type, as in Figure 4.5. We should mention that this time we consider *Store*′ as well as the powerset cpo as discrete cpo's, in order to conform to the semantics of ML-like languages. For a typing environment Γ we define $\mathcal{V}[\![\Gamma]\!] = \prod_{x\in\mathsf{dom}\,\Gamma} \mathcal{V}[\![\Gamma(x)]\!]$.

We define semantic functions for each syntactic category. The meaning of a well-typed term $\Gamma \vdash_\bullet t : \tau$, where $\bullet$ ranges over C, V, E, P is given as follows:

$$
\begin{array}{lcl}
[\![t]\!]^P & : & \mathit{Store}'_\perp \\
[\![t]\!]^C & : & \mathcal{V}[\![\Gamma]\!] \to_c \mathit{Store}' \to_c (\mathcal{V}[\![\tau]\!] \times \mathit{Store}')_\perp \\
[\![t]\!]^E & : & \mathcal{V}[\![\Gamma]\!] \to_c \mathit{Store}' \to_c \mathcal{V}[\![\tau]\!]_\perp \\
[\![t]\!]^V & : & \mathcal{V}[\![\Gamma]\!] \to_c \mathcal{V}[\![\tau]\!]
\end{array}
$$

The semantics of the language constructs are given in Figure 4.6.

The semantics of a constant $\Gamma \vdash_V c : \tau$ is based on an interpretation $\mathcal{I}(c) : \mathcal{V}[\![\tau]\!]$, which is given in Figure 4.7. Function Acc^1 is essentially the same function as *Acc*, just modified to account for the fact that the iterated function returns now an element of $(\mathcal{V}[\![unit]\!] \times \mathit{Store}')_\perp$ instead of *Store*′. The conversion should be obvious so we do not go into details.

As expected, it can be proven that the semantics is well-defined. Indeed, all the defined domains are cpo's, and it can be proven that the interpretation of the constants gives continuous functions.

`UpdateNRho`	:	$(\mathit{VAR\ list} \times \mathit{LPROC\ set\ list}) \to \mathit{unit}$
`UpdateRho`	:	$(\mathit{VAR} \times \mathit{LPROC\ set}) \to \mathit{unit}$
`UpdateDelta`	:	$(\mathit{LAB} \times \mathit{LPROC\ set}) \to \mathit{unit}$
`Ahat`	:	$\mathit{ARGFUN} \to \mathit{LPROC\ set}$
`AhatN`	:	$\mathit{ARGFUN\ list} \to \mathit{LPROC\ set\ list}$
`lookup`	:	$\mathit{VAR} \to \mathit{LPROC\ set}$
`Set.Fmap`	:	$(((\mathit{PROC} \times \mathit{LPROC\ set\ list}) \to \mathit{unit})$ $\times \mathit{LPROC\ set} \times \mathit{LPROC\ set\ list}) \to \mathit{unit}$
`List.nth`	:	$(\mathit{LPROC\ set\ list} \times \mathit{nat}) \to \mathit{LPROC\ set}$

`Older`	:	$\mathit{LAB} \to \mathit{bool}$	`Set.Single`	:	$\mathit{LPROC} \to \mathit{LPROC\ set}$
`Mark`	:	$\mathit{LAB} \to \mathit{unit}$	`Set.Empty`	:	$\mathit{LPROC\ set}$
`Tlab,Flab`	:	$\mathit{LAB} \to \mathit{LAB}$	`Start`	:	$\mathit{LPROC\ set\ list}$
`Plab`	:	$\mathit{LAB} \to \mathit{LAB}$	`nil`	:	$\mathit{LPROC\ set\ list}$

Figure 4.3: IMP predefined constants

$$\frac{\begin{array}{l}\vdots\\ [\ldots f_k \mapsto ((\beta_{k1}\times\ldots\times\beta_{ka_k})\to \mathit{unit}),\ldots,x_{i1}\mapsto\beta_{i1},\ldots,x_{ia_i}\mapsto\beta_{ia_i}]\vdash_C C_i:\mathit{unit}\\ \vdots\\ [\ldots f_k\mapsto((\beta_{k1}\times\ldots\times\beta_{ka_k})\to\mathit{unit}),\ldots]\vdash_C C:\mathit{unit}\end{array}}{\texttt{letrec}\ \ldots\texttt{fun}\ f\texttt{=}_i\texttt{(}x_{i1}^{\beta_{i1}},\ldots,x_{ia_i}^{\beta_{ia_i}}\texttt{)=}C_i\ldots\ \texttt{in}\ C:\mathit{unit}}$$

$$\frac{\Gamma\vdash_V V:(\tau_1\times\ldots\times\tau_n)\to\mathit{unit}\quad \ldots\ \Gamma\vdash_E E_i:\tau_i\ \ldots}{\Gamma\vdash_C V\texttt{(}E_1\texttt{,}\ldots\texttt{,}E_n\texttt{)}:\mathit{unit}}\qquad\frac{}{\Gamma\vdash_C \texttt{skip}:\mathit{unit}}$$

$$\frac{\Gamma\vdash_E E:\mathit{bool}\quad \Gamma\vdash_C C_1:\mathit{unit}\quad \Gamma\vdash_C C_2:\mathit{unit}}{\Gamma\vdash_C \texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2:\mathit{unit}}\qquad\frac{\ldots\ \Gamma\vdash_C C_i:\mathit{unit}\ \ldots}{\Gamma\vdash_C \texttt{(}C_1\texttt{;}\ldots\texttt{;}C_n\texttt{)}:\mathit{unit}}$$

$$\frac{\begin{array}{l}\Gamma\vdash_V V:\mathit{CALL}\\ \Gamma[e\mapsto\mathit{ARGFUN},es\mapsto\mathit{ARGFUN\ list},l_1\mapsto\mathit{LAB}]\vdash_C C_1:\mathit{unit}\\ \Gamma[ids\mapsto\mathit{VAR\ list},es\mapsto\mathit{ARGFUN\ list},e\mapsto\mathit{CALL},l_2\mapsto\mathit{LAB}]\vdash_C C_2:\mathit{unit}\end{array}}{\begin{array}{l}\Gamma\vdash_C\ \texttt{caseCall}\ V\ \texttt{of}\ l_1\texttt{:APP(}e,es\texttt{)=>}\ C_1\quad:\mathit{unit}\\ \qquad\qquad\texttt{|}l_2\texttt{:REC(}ids,es,e\texttt{)=>}\ C_2\end{array}}$$

$$\frac{\begin{array}{l}\Gamma\vdash_V V:\mathit{PROC}\\ \Gamma[ids\mapsto\mathit{VAR\ list},e\mapsto\mathit{CALL},l_1\mapsto\mathit{LAB}]\vdash_C C_1:\mathit{unit}\\ \Gamma[l_2\mapsto\mathit{LAB}]\vdash_C C_2:\mathit{unit}\\ \Gamma[l_3\mapsto\mathit{LAB}]\vdash_C C_3:\mathit{unit}\\ \Gamma\vdash_C C_4:\mathit{unit}\end{array}}{\begin{array}{l}\Gamma\vdash_C\ \texttt{caseProc}\ V\ \texttt{of}\ l_1\texttt{:LAM(}ids,e\texttt{)=>}\ C_1\quad:\mathit{unit}\\ \qquad\qquad\texttt{|}l_2\texttt{:BIF =>}\ C_2\\ \qquad\qquad\texttt{|}l_3\texttt{:BPLUS =>}\ C_3\\ \qquad\qquad\texttt{|Stop =>}\ C_4\end{array}}\qquad\frac{\Gamma(x)=\tau}{\Gamma\vdash_V x:\tau}\qquad\frac{\Gamma\vdash_V v:\tau}{\Gamma\vdash_E v:\tau}$$

$$\frac{\Gamma\vdash_V V:(\tau_1\times\ldots\times\tau_n)\to\tau\quad\ldots\ \Gamma\vdash_V V_i:\tau_i\ \ldots}{\Gamma\vdash_E V\texttt{(}V_1\texttt{,}\ldots\texttt{,}V_n\texttt{)}:\tau}\qquad\frac{\ldots\ \Gamma\vdash_E E_i:b\ \ldots}{\Gamma\vdash_E \texttt{[}E_1\texttt{,}\ldots\texttt{,}E_n\texttt{]}:b\ \mathit{list}}$$

Figure 4.4: Typing rules for IMP

$$
\begin{array}{rclrcl}
\mathcal{V}[\![\textsf{nat}]\!] & = & \mathbf{N} & \mathcal{V}[\![\textsf{LPROC set}]\!] & = & \mathcal{P}(\mathit{LPROC}) \\
\mathcal{V}[\![\textsf{bool}]\!] & = & \mathit{bool} & \mathcal{V}[\![\textsf{ARGFUN}]\!] & = & \mathit{ARG} \cup \mathit{FUN} \\
\mathcal{V}[\![\textsf{unit}]\!] & = & \{*\} & \mathcal{V}[\![\textsf{VAR}]\!] & = & \mathit{VAR} \\
\mathcal{V}[\![\textsf{PROC}]\!] & = & \mathit{PROC} & \mathcal{V}[\![\textsf{CALL}]\!] & = & \mathit{CALL} \\
\mathcal{V}[\![\textsf{LPROC}]\!] & = & \mathit{LPROC} & \mathcal{V}[\![\textsf{LAB}]\!] & = & \mathit{LAB}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{V}[\![b\ \textsf{list}]\!] & = & \mathcal{V}[\![b]\!]^* \\
\mathcal{V}[\![(\tau_1 \times ... \times \tau_n)]\!] & = & (\mathcal{V}[\![\tau_1]\!] \times ... \times \mathcal{V}[\![\tau_n]\!]) \\
\mathcal{V}[\![\tau_1 \to \tau_2]\!] & = & \mathcal{V}[\![\tau_1]\!] \to_c \mathit{Store}' \to_c (\mathcal{V}[\![\tau_2]\!] \times \mathit{Store}')_\perp
\end{array}
$$

Figure 4.5: Interpretation of types

$$
\begin{array}{rcl}
[\![x]\!]^V e & = & e(x) \\
[\![c]\!]^V e & = & \mathcal{I}(x) \\
\\
[\![V\texttt{(}V_1, ..., V_n\texttt{)}]\!]^E e\sigma & = & (\lambda(v,\sigma).\mathsf{up}(v))^\dagger(([\![V]\!]^V e)([\![V_1]\!]^V e, ..., [\![V_n]\!]^V e)\sigma) \\
[\![\texttt{[}E_1, ..., E_n\texttt{]}]\!]^E e\sigma & = & (\lambda v_1.....(\lambda v_n.\mathsf{up}(\langle x_1, ..., x_n\rangle))^\dagger([\![E_n]\!]^E e\sigma)...)^\dagger([\![E_1]\!]^E \sigma) \\
[\![V]\!]^E e\sigma & = & \mathsf{up}([\![V]\!]^V e) \\
\\
[\![V\texttt{(}E_1, ..., E_n\texttt{)}]\!]^C e\sigma & = & (\lambda x_1.(...(\lambda x_n.(([\![V]\!]^V e)(x_1, ..., x_n)\sigma))^\dagger [\![E_n]\!]^E e\sigma... \\
 & & \qquad ...)^\dagger [\![E_1]\!]^E e\sigma \\
[\![\texttt{if } E_0 \texttt{ then } C_1 \texttt{ else } C_2]\!]^C e\sigma & = & (\lambda b.\mathbf{if}\ b\ \mathbf{then}\ [\![C_1]\!]^C e\sigma\ \mathbf{else}\ [\![C_2]\!]^C e\sigma)^\dagger([\![E_0]\!]^E e\sigma) \\
[\![\texttt{skip}]\!]^C e\sigma & = & \mathsf{up}(\emptyset, \sigma) \\
[\![C_1; ...; C_n]\!]^C e\sigma & = & (\lambda(v_n, \sigma_n).[\![C_n]\!]^C e\sigma_n)^\dagger...([\![C_1]\!]^C e\sigma)
\end{array}
$$

$$
[\![\texttt{letrec } ...\texttt{fun } f\texttt{=}_i\texttt{(}x_{i1}, ...x_{in}\texttt{)=}C_i...\ \texttt{ in } C]\!]^P = (\lambda(v,\sigma').\mathsf{up}(\sigma'))^\dagger [\![C]\!]^C [f_i \mapsto \phi_i]_i(\emptyset, \emptyset, 0_C, 0)
$$

$$
\text{where } (\phi_1, ..., \phi_n) = \mathit{Fix}(\lambda(\phi_1, ..., \phi_n).\ (...\lambda(v_{i1}, ..., v_{in}).\lambda\sigma'.[\![C_i]\!]^C(e[f_j \mapsto \phi_j]_j[x_{ik} \mapsto v_{ik}]_k)\sigma'...))
$$

$$
\left[\begin{array}{l}
\texttt{caseCall } V \texttt{ of } l_1\texttt{:APP(}e_0, es\texttt{)=> } C_1 \\
\quad \texttt{| } l_2\texttt{:REC(}ids, es, e_1\texttt{)=>}C_2
\end{array}\right] e\sigma = (\phi)^\dagger([\![V]\!]^V e)
$$

$$
\text{where } \phi = \lambda x. \begin{cases}
[\![C_1]\!]^C e'\sigma & \text{if } x = c\text{:}\texttt{(}f\ \ a_1\ ...\ a_n\texttt{)} \\
 & \text{where } e' = e\ \ [l_1 \mapsto c, e_0 \mapsto f, es \mapsto \langle a_1, ..., a_n\rangle] \\
[\![C_2]\!]^C e'\sigma & \text{if } x = c\text{:}\texttt{(letrec ((}x_1\ d_1\texttt{)...)}\ \ e_2\texttt{)} \\
 & \text{where } e' = e\ \ [l_2 \mapsto c, e_1 \mapsto e_2, ids \mapsto \langle x_1, ..., x_n\rangle, \\
 & \qquad\qquad es \mapsto \langle d_1, ..., d_n\rangle]
\end{cases}
$$

$$
\left[\begin{array}{l}
\texttt{caseProc } V \texttt{ of } l_1\texttt{:LAM(}ids, e_0\texttt{)=> } C_1 \\
\quad \texttt{| } l_2\texttt{:BIF => } C_2 \\
\quad \texttt{| } l_3\texttt{:BPLUS => } C_3 \\
\quad \texttt{| Stop =>}C_4
\end{array}\right] e\sigma = (\phi)^\dagger([\![V]\!]^V e)
$$

$$
\text{where } \phi = \lambda x. \begin{cases}
[\![C_1]\!]^C e'\sigma & \text{if } x = l\text{:}\texttt{(}\lambda\ \texttt{(}id_1\ ...\ id_n\texttt{)}\ c\texttt{)} \\
 & \text{where } e' = e\ \ [l_1 \mapsto l, e_0 \mapsto c, ids \mapsto \langle id_1\ ...\ id_n\rangle] \\
[\![C_2]\!]^C e[l_2 \mapsto l]\sigma & \text{if } x = l\text{:}\texttt{if} \\
[\![C_3]\!]^C e[l_3 \mapsto l]\sigma & \text{if } x = l\text{:}\texttt{+} \\
[\![C_4]\!]^C e\sigma & \text{if } x = \mathit{Stop}
\end{cases}
$$

Figure 4.6: Semantics of IMP language constructs

$$
\begin{array}{rcl}
\mathcal{I}(\texttt{UpdateNRho}) & = & \lambda(ids, Sets).\lambda\sigma \begin{cases} \mathsf{up}(\emptyset, \textit{UnionN}(\langle(id_1, S_1), .., (id_n, S_n)\rangle)\sigma) \\ \text{if} \quad ids = \langle id_1, ..., id_n\rangle, Sets = \langle S_1, ..., S_m\rangle \\ \quad\text{and } m = n \\ \bot \text{ otherwise} \end{cases} \\
\mathcal{I}(\texttt{UpdateRho}) & = & \lambda(id, S).\lambda\sigma.\mathsf{up}(\emptyset, \textit{Union}([id \mapsto S])\sigma) \\
\mathcal{I}(\texttt{UpdateDelta}) & = & \lambda(lab, S).\lambda\big(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t\big).\mathsf{up}(\emptyset, (\widehat{\rho}, [lab \mapsto S] \sqcup \widehat{\delta}, \widehat{\tau}, t)) \\
\mathcal{I}(\texttt{Ahat}) & = & \lambda x.\lambda\sigma.\mathsf{up}(\emptyset, \widehat{\mathcal{A}}'(x)\sigma) \\
\mathcal{I}(\texttt{AhatN}) & = & \lambda\langle E_1, ..., E_n\rangle.\lambda\sigma.\mathsf{up}(\langle\widehat{\mathcal{A}}'(E_1)\sigma, ..., \widehat{\mathcal{A}}'(E_n)\sigma\rangle, \sigma) \\
\mathcal{I}(\texttt{lookup}) & = & \lambda x.\lambda\sigma.\mathsf{up}(\sigma(x), \sigma) \\
\mathcal{I}(\texttt{Set.Fmap}) & = & \lambda(F, S, p).\lambda\sigma.((\lambda\sigma'.\mathsf{up}(\emptyset, \sigma'))^{\dagger}(Acc^1(F, S, p)\sigma)) \\
\mathcal{I}(\texttt{List.nth}) & = & \lambda(l, n).\lambda\sigma. \begin{cases} \mathsf{up}(A_n, \sigma) & \text{if } l = \langle A_0, ..., A_m\rangle \wedge n \leq m \\ \bot & \text{otherwise} \end{cases} \\
\mathcal{I}(\texttt{Older}) & = & \lambda l.\lambda\big(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t\big).\mathsf{up}(\widehat{\tau}(\ulcorner l \urcorner) < t, (\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t)) \\
\mathcal{I}(\texttt{Mark}) & = & \lambda l.\lambda\big(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t\big).\mathsf{up}(\emptyset, (\widehat{\rho}, \widehat{\delta}, \widehat{\tau}[\ulcorner l \urcorner \mapsto t], t))
\end{array}
$$

$$
\begin{array}{rclrcl}
\mathcal{I}(\texttt{Set.Single}) & = & \lambda v.\lambda\sigma.\mathsf{up}(\{v\}, \sigma) & \mathcal{I}(\texttt{Set.Empty}) & = & \{\emptyset\} \\
\mathcal{I}(\texttt{Tlab}) & = & \lambda l.\lambda\sigma.\mathsf{up}(\texttt{if}^t_l, \sigma) & \mathcal{I}(\texttt{Start}) & = & \langle \textit{Stop}\rangle \\
\mathcal{I}(\texttt{Flab}) & = & \lambda l.\lambda\sigma.\mathsf{up}(\texttt{if}^f_l, \sigma) & \mathcal{I}(\texttt{nil}) & = & \langle\rangle \\
\mathcal{I}(\texttt{Plab}) & = & \lambda l.\lambda\sigma.\mathsf{up}(\texttt{+}_l, \sigma) & \mathcal{I}(\texttt{Prg}) & = & \textit{Prg}
\end{array}
$$

Figure 4.7: Interpretation of IMP constants

4.4 Implementation of 0-CFA

We present the program *ICFA* from Figure 4.8. It can be easily seen that it type-checks and thus we do not go into details. We prove that the program is computing the state-passing 0-CFA for the program *Prg*, as given in Chapter 3.

Theorem 4.4.1 (Correctness of implementation).

$$[\![ICFA]\!]^P = \textit{up}(\widehat{\mathcal{F}}' \;\; P \;\; \langle\{\textit{Stop}\}\rangle \;\; (\emptyset, \emptyset, 0_{CALL}, 0))$$

Before going into the proof, we can have a closer look at the program. Let `CH` be the body of the definition of the function `Chat`, and similarly `FH` for `Fhat`. Let H'' be the iterated function in the fixed point given in the denotation of *ICFA*:

$$
\begin{array}{rl}
H'' = \lambda(\widehat{\mathcal{F}}'', \widehat{\mathcal{C}}''). & (\lambda e_1.\lambda\sigma'.[\![\texttt{CH}]\!]^C e[\texttt{exp} \mapsto e_1]\sigma', \\
& \lambda(e_2, v).\lambda\sigma'.[\![\texttt{FH}]\!]^C e[\texttt{exp} \mapsto e_2, \texttt{Vals} \mapsto v]\sigma') \\
& \text{where } e = [\texttt{Fhat} \mapsto \widehat{\mathcal{F}}'', \texttt{Chat} \mapsto \widehat{\mathcal{C}}'']
\end{array}
$$

The least fixed point of H'' is obtained by recursive application of the H'' to the bottom value, in our case, constructing a chain:

$$
\begin{array}{rcl}
\big(\widehat{\mathcal{C}}''_i, \widehat{\mathcal{F}}''_i\big)_{i \in \mathbf{N}} & \in & ((CALL \to Store' \to (unit \times Store')_{\bot}) \times \\
& & \quad ((PROC \times \mathcal{P}(LPROC)^*) \to Store' \to (unit \times Store'))) \\
\big(\widehat{\mathcal{C}}''_0, \widehat{\mathcal{F}}''_0\big) & = & (\lambda c.\lambda\sigma.\bot, \lambda(f, L).\lambda\sigma.\bot) \\
\big(\widehat{\mathcal{C}}''_{i+1}, \widehat{\mathcal{F}}''_{i+1}\big) & = & H''\big(\widehat{\mathcal{C}}''_i, \widehat{\mathcal{F}}''_i\big)
\end{array}
$$

$(\lambda e.\lambda\sigma.\bot, \lambda(f, L).\lambda\sigma.\bot)$. Similarly, recalling H' from Chapter 3, the least fixed point of H' is also obtained by recursive application

$$
\begin{array}{rcl}
\big(\widehat{\mathcal{C}}'_i, \widehat{\mathcal{F}}'_i\big)_{i \in \mathbf{N}} & \in & (CSPACE' \times FSPACE') \\
\big(\widehat{\mathcal{C}}'_0, \widehat{\mathcal{F}}'_0\big) & = & (\lambda c.\lambda\sigma.(\emptyset, \emptyset, 0_{CALL}, 0), \lambda(f, L).\lambda\sigma.(\emptyset, \emptyset, 0_{CALL}, 0)) \\
\big(\widehat{\mathcal{C}}'_{i+1}, \widehat{\mathcal{F}}'_{i+1}\big) & = & H'\big(\widehat{\mathcal{C}}'_i, \widehat{\mathcal{F}}'_i\big)
\end{array}
$$

We prove that

```
letrec
    fun Chat(exp) =
        caseCall exp of
             lab:APP(f,exs) =>
                 if Older(lab) then
                     (Mark(lab);
                      UpdateDelta(lab,Ahat(f));
                      Set.Fmap(Fhat,Ahat(f),AhatN(exs)))
                 else skip
           | lab:REC(ids,funs,e) =>
                     (UpdateNRho(ids,AhatN(funs));
                      Chat(e))
    fun Fhat(exp,Vals) =
        caseProc exp of
             lab:LAM(ids,e) =>
                 (UpdateNRho(ids,Vals);
                  Chat(e))
           | lab:BIF =>
                 (UpdateDelta(Tlab(lab),List.nth(Vals,1));
                  UpdateDelta(Flab(lab),List.nth(Vals,2));
                  Set.Fmap(Fhat,List.nth(Vals,1),nil);
                  Set.Fmap(Fhat,List.nth(Vals,2),nil))
           | lab:BPLUS =>
                 (UpdateDelta(Plab(lab),List.nth(Vals,2));
                  Set.Fmap(Fhat,List.nth(Vals,2),nil))
           | Stop => skip
in
    Fhat(Prg,Start)
end
```

Figure 4.8: *ICFA*: program for computing the 0-CFA

Lemma 4.4.2. *For all $i \in \mathbf{N}$ the following holds:*

- $\forall c \in CALL, \sigma \in Store'$

$$\widehat{\mathcal{C}}''_i\ c\ \sigma = \bot \vee \widehat{\mathcal{C}}''_i\ c\ \sigma = up(\emptyset, \widehat{\mathcal{C}}'_i\ c\ \sigma)$$

- $\forall f \in PROC, A_1, ..., A_n \in \mathcal{P}(LPROC).(f, \langle A_1, ..., A_n\rangle) \in RFPAR', \forall \sigma \in Store'$

$$\widehat{\mathcal{F}}''_i\ f\ \langle A_1, ..., A_n\rangle\ \sigma = \bot \vee \widehat{\mathcal{F}}''_i\ f\ \langle A_1, ..., A_n\rangle\ \sigma = up(\emptyset, \widehat{\mathcal{F}}'_i\ f\ A_1, ..., A_n\ \sigma)$$

Proof. The lemma is proven by induction. The base case holds trivially. Assuming the statement holds for $i \in \mathbf{N}$, we prove that it holds for $i + 1$.

Take $C \equiv c{:}(f\ a_1\ ...\ a_n) \in CALL$, $(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t) \in Store'$. By definition, we have:

$$\widehat{\mathcal{C}}''_{i+1}C(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t) = [\![\mathtt{CH}]\!]^C[\mathtt{Chat} \mapsto \widehat{\mathcal{C}}''_i, \mathtt{Fhat} \mapsto \widehat{\mathcal{F}}''_i, \mathtt{exp} \mapsto C](\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t)$$

Let

$$e = [\mathtt{Chat} \mapsto \widehat{\mathcal{C}}''_i, \mathtt{Fhat} \mapsto \widehat{\mathcal{F}}''_i, \mathtt{exp} \mapsto C]$$
$$e' = e[\mathtt{lab} \mapsto c, \mathtt{f} \mapsto f, \mathtt{exs} \mapsto \langle a_1, ..., a_n\rangle]$$

If we assume $\widehat{\tau}(c) < t$ we obtain:

$$\begin{aligned}
\widehat{\mathcal{C}}''_{i+1}C(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t) &= [\![\mathtt{if\ Older(lab)\ then...}]\!]^C e'(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t) \\
[\![\mathtt{Older(lab)}]\!]^E e'(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t) &= \mathtt{up}(\widehat{\tau}(c) < t) \\
[\![\mathtt{Mark(lab)}]\!]^C e'(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t) &= \mathtt{up}(\emptyset, (\widehat{\rho}, \widehat{\delta}, \widehat{\tau}[c \mapsto t], t)) \\
[\![\mathtt{UpdateDelta(lab,Ahat(f))}]\!]^C e'(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}[c \mapsto t], t) &= \mathtt{up}(\emptyset, (\widehat{\rho}, \widehat{\delta} \sqcup [c \mapsto F], \widehat{\tau}[c \mapsto t], t))
\end{aligned}$$

where $F = \widehat{\mathcal{A}}'[\![f]\!]\widehat{\rho}$. Let $\big(\widehat{\rho}_1, \widehat{\delta}_1, \widehat{\tau}_1, t_1\big) = \big(\widehat{\rho}, \widehat{\delta} \sqcup [c \mapsto F], \widehat{\tau}[c \mapsto t], t\big)$ and $B_i = \widehat{\mathcal{A}}'[\![f]\!]\widehat{\rho}$. We obtain:

$$\begin{aligned}&[\![\texttt{Set.Fmap(Fhat,Ahat(f),AhatN(exs))}]\!]^C e'\big(\widehat{\rho}_1, \widehat{\delta}_1, \widehat{\tau}_1, t_1\big) = \\ &= (\lambda\sigma'.\mathsf{up}(\emptyset, \sigma'))^\dagger(Acc^1(\widehat{\mathcal{F}}''_i, F, \langle B_1, ..., B_n\rangle)\big(\widehat{\rho}_1, \widehat{\delta}_1, \widehat{\tau}_1, t_1\big))\end{aligned}$$

And then:

$$\widehat{\mathcal{C}}''_{i+1}c\big(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t\big) = (\lambda\sigma'.\mathsf{up}(\emptyset, \sigma'))^\dagger(Acc^1(\widehat{\mathcal{F}}''_i, F, \langle B_1, ..., B_n\rangle)\big(\widehat{\rho}_1, \widehat{\delta}_1, \widehat{\tau}_1, t_1\big))$$

If one of the $\widehat{\mathcal{F}}''_i\ f_i\ \langle B_1, ..., B_n\rangle\ \big(\widehat{\rho}_k, \widehat{\delta}_k, \widehat{\tau}_k, t_k\big)$ involved in the unfolding of Acc^1 is $\bot$ then the above is also $\bot$. Otherwise, by the induction hypothesis we can prove:

$$Acc^1(\widehat{\mathcal{F}}''_i, F, \langle B_1, ..., B_n\rangle)\big(\widehat{\rho}_1, \widehat{\delta}_1, \widehat{\tau}_1, t_1\big) = Acc(\widehat{\mathcal{F}}'_i, F, \langle B_1, ..., B_n\rangle)\big(\widehat{\rho}, \widehat{\delta}, \widehat{\tau}, t\big)$$

which gives us that:

$$\widehat{\mathcal{C}}''_i\ c\ \sigma = \mathsf{up}(\emptyset, \widehat{\mathcal{C}}'_i\ c\ \sigma)$$

In the case when $\widehat{\tau}(c) = t$ then we obtain:

$$\widehat{\mathcal{C}}''_i\ c\ \sigma = [\![\texttt{skip}]\!]^C e'\sigma = \sigma = \mathsf{up}(\emptyset, \widehat{\mathcal{C}}'_i\ c\ \sigma)$$

The rest of the cases can be proven similarly. □

Being taken from finite domains, the limits of the two chains, namely the least fixed points of H' and H'' will also have the property from the lemma. Let $(\widehat{\mathcal{C}}^I, \widehat{\mathcal{F}}^I) = Fix(H'')$. The above yields

Corollary 4.4.3.

$$\begin{aligned}&\widehat{\mathcal{F}}^I(Prg, \langle\{Stop\}\rangle)\big(\emptyset, \emptyset, 0_{CALL}, 0\big) = \bot\ \textit{or} \\ &\widehat{\mathcal{F}}^I(Prg, \langle\{Stop\}\rangle)\big(\emptyset, \emptyset, 0_{CALL}, 0\big) = \mathsf{up}(\emptyset, \widehat{\mathcal{F}}'\ Prg\ \langle\{Stop\}\rangle\ \big(\emptyset, \emptyset, 0_{CALL}, 0\big))\end{aligned}$$

To prove the theorem we need to prove that:

$$\widehat{\mathcal{F}}^I(Prg, \langle\{Stop\}\rangle)\big(\emptyset, \emptyset, 0_{CALL}, 0\big) \neq \bot$$

We prove a stronger result:

Lemma 4.4.4. *$\forall\sigma \in Store'$ valid, for all $f \in PROC$ and $A_1, ..., A_n \in \mathcal{P}(LPROC)$ such that $(f, \langle A_1, ..., A_n\rangle) \in RFPAR'$ and $\ulcorner A_i \urcorner \in \mathcal{P}(LAM)$*

$$\widehat{\mathcal{F}}^I(P, \langle A_1, ..., A_n\rangle)\sigma \neq \bot$$

Proof. We prove the lemma by well-founded induction. We recall the well-founded ordering on *Store'* introduced in Definition 3.3.5. Taking Q as a predicate on σ, we prove that

$$\forall\sigma \in Store'.(\forall\sigma' \prec \sigma.Q(\sigma')) \Rightarrow Q(\sigma)$$

The proof goes very much the same as the proof of Lemma 3.3.4.

Consider an arbitrary valid σ. Assume $(\forall\sigma' \prec \sigma.Q(\sigma'))$. We prove $Q(\sigma)$. Take first $f \in LAM$ and $A_1, ..., A_n \in LPROC$ such that $\ulcorner A_i \urcorner \in \mathcal{P}(LAM)$ and $(f, \langle A_1, ..., A_n\rangle) \in RFPAR'$. Assuming $f \equiv l$:`(`λ `(`x_1 ... x_n`)` e`)`, we can see from the fixed point property and definition of the semantics that:

$$\widehat{\mathcal{F}}^I(f, \langle A_1, ..., A_n\rangle)\sigma = \widehat{\mathcal{C}}^I\ e\ \sigma_1$$

where $\sigma_1 \prec \sigma$ or $\sigma_1 = \sigma$. We do now a case analysis on e.

Assume e is of the form c:`(`$f\ a_1$ ... a_n`)`. Then:

- If $\widehat{\tau}_1(e) = t_1$ then $Q(\sigma)$ holds trivially.
- If $\widehat{\tau}_1(e) < t_1$ then we obtain

  $$(\lambda\sigma'.\mathtt{up}(\emptyset, \sigma'))^\dagger(Acc^1(\widehat{\mathcal{F}}^I, F, \langle B_1, ..., B_n\rangle)\sigma_2)$$

  where we can prove that $\ulcorner B_i \urcorner \in \mathcal{P}(LAM)$ and that $\sigma_2 \prec \sigma_1$ and σ_2 valid. By directly using the hypothesis we easily prove that the above yields a non-$\perp$ value.

In the other case of e, we need to construct a recursive judgment on the structure of e, which ends with a prove as above.

From the above we can now easily prove that the result holds also for $f \in PRIM$, similar as in Lemma 3.3.4. The case $f = Stop$ is again trivial. □

This leads us to:

Proof of Theorem 4.4.1. We have:

$$[\![ICFA]\!]^P = (\lambda(v, \sigma').\sigma')^\dagger([\![\mathtt{Fhat(Prg,Start)}]\!]^C[\mathtt{Chat} \mapsto \widehat{\mathcal{C}}^I, \mathtt{Fhat} \mapsto \widehat{\mathcal{F}}^I](\emptyset, \emptyset, 0_{CALL}, 0)) =$$
$$= (\lambda(v, \sigma').\sigma')^\dagger(\widehat{\mathcal{F}}^I(Prg, \langle\{Stop\}\rangle)(\emptyset, \emptyset, 0_{CALL}, 0))$$

From Corollary 4.4.3 and Lemma 4.4.4:

$$\widehat{\mathcal{F}}^I(Prg, \langle\{Stop\}\rangle)(\emptyset, \emptyset, 0_{CALL}, 0) = \mathtt{up}(\emptyset, \widehat{\mathcal{F}}'\ Prg\ \langle\{Stop\}\rangle\ (\emptyset, \emptyset, 0_{CALL}, 0))$$

which yields the statement of the theorem. □

4.5 Summary and Conclusions

We have formalized an implementation of the modified 0-CFA analysis given in Chapter 3. This has been achieved by describing the formal semantics of the implementation language IMP, by presenting the program *ICFA* in IMP and proving that its meaning in the formal semantics is equal with the result of the 0-CFA analysis.

The analysis in the form of the *ICFA* program is the subject of our further development. Using the formal semantics of IMP, we are able to prove the correctness of the transformations we perform on *ICFA* in order to specialize it with respect to the source program *Prg*.

Chapter 5

Partial-Evaluation Steps in IMP

5.1 Introduction

Our goal is to specialize the program *ICFA* with respect to *Prg*. In a run of *ICFA*, we repeatedly perform case instructions over the same syntactic constructs of *Prg*, which is an overhead that could be removed by specialization. Another overhead we identify is incurred by manipulation of lists. Since the lists manipulated in the program are constructed from the syntax of the input program, we can consider it as overhead induced by the syntax of the source program.

Considering a standard binding-time separation [9] on *ICFA*, our constraints would be that case instructions should be static and the store operations (unions, time-stamps tests) should be dynamic.

The main problem with this approach is that in this case, the control flow of the *ICFA* program is dynamic. Considering a monovariant binding-time annotation [9] of `Fhat`, we are forced to declare all its parameters dynamic, and then much of the case instructions are still performed by the residual program.

A solution for a satisfactory specialization can be found though. As we show, we are able to remove all case instructions and construction of lists, using previously non-formalized methods: exploiting bounded static variation [9] and performing arity raising. Our solution is to produce a specialized version of the program *ICFA*, by performing a series of meaning preserving program transformations driven by the source program, thus giving a formal proof for our non-trivial specializer.

In the transformation process, we make explicit the polyvariant specialization of the `Fhat` function. For this purpose, we add a new construct to the language IMP. Considering that $\mathtt{l}_1, ..., \mathtt{l}_\Psi$ are the elements of $PROC$, we define

$$\begin{array}{l}\mathtt{caseX}\ V\ \mathtt{of} \\ \quad \mathtt{l}_1\ \mathtt{=>}\ C_1 \\ \quad \vdots \\ \mathtt{|}\ \mathtt{l}_\Psi\ \mathtt{=>}\ C_\Psi\end{array}$$

to be a program construct doing a case analysis on the value of an expression of type $PROC$. The typing rule for the construct and its semantics are given in Figure 5.1. The new construct amounts to what is known as “The Trick” [9], and it is strictly dependent on the input program *Prg*. However, we will use it only for formalization purposes.

5.2 Program Equivalence

Our formal development of specialization is relying on a notion of program equivalence:

$$\frac{\begin{array}{l}\Gamma \vdash_V V : \mathit{PROC} \\ \Gamma \vdash_C C_1 : \mathit{unit} \\ \quad\vdots \\ \Gamma \vdash_C C_\Psi : \mathit{unit}\end{array}}{\Gamma \vdash_C \begin{array}{l}\texttt{caseX}\ V\ \texttt{of} \\ \quad \texttt{l}_1\ \texttt{=>}\ C_1 \\ \quad\vdots \\ \texttt{|}\ \texttt{l}_\Psi\ \texttt{=>}\ C_\Psi\end{array} : \mathit{unit}} \qquad \left[\!\!\left[\begin{array}{l}\texttt{caseX}\ V\ \texttt{of} \\ \quad \texttt{l}_1\ \texttt{=>}\ C_1 \\ \quad\vdots \\ \texttt{|}\ \texttt{l}_\Psi\ \texttt{=>}\ C_\Psi\end{array}\right]\!\!\right] e\sigma = (\phi)^\dagger(\llbracket V \rrbracket^V e)$$

$$\text{with } \phi(x) = \begin{cases} \llbracket C_1 \rrbracket e\sigma & \text{if } x = \texttt{l}_1 \\ \quad\vdots \\ \llbracket C_\Psi \rrbracket e\sigma & \text{if } x = \texttt{l}_\Psi \end{cases}$$

Figure 5.1: Typing rule and semantics for the `caseX` construct

Definition 5.2.1. *For $\Gamma \vdash_\bullet T_1 : \tau$ and $\Gamma \vdash_\bullet T_2 : \tau$ two well-typed IMP terms of the same type, we say that they are equivalent ($T_1 \equiv T_2$) iff $\llbracket T_1 \rrbracket^\bullet = \llbracket T_2 \rrbracket^\bullet$.*

We can prove by structural induction, using the compositionality of our semantics, that the equivalence of two terms implies contextual equivalence, i.e. replacing a sub-term of a program with an equivalent term does not change the result of evaluating the program.

5.3 Substitution Lemmas

For a term P, we consider $T[T_1/x]$ to denote a capture-avoiding substitution of variable x with the term T_1 inside T. We can prove by structural induction the following substitution lemmas.

Lemma 5.3.1 (Value substitution). *Let T be a term and x a variable such that $\Gamma[x \mapsto \tau] \vdash T : \tau_1$, for some τ and τ_1, and let $\texttt{c} : \tau$ be a constant. Then*

$$\llbracket T \rrbracket^\bullet e[x \mapsto \mathcal{I}(\texttt{c})] = \llbracket T[\texttt{c}/x] \rrbracket^\bullet e$$

Lemma 5.3.2 (Variable substitution). *Let T be a term and x a variable such that $\Gamma[x \mapsto \tau] \vdash T : \tau_1$ for some τ and τ_1. Let $v \in \mathcal{V}\llbracket \tau \rrbracket$ and y be a fresh variable. Then*

$$\llbracket T \rrbracket^\bullet e[x \mapsto v] = \llbracket T[y/x] \rrbracket^\bullet e[y \mapsto v]$$

Lemma 5.3.3. *Let T be a term and x a variable such that $\Gamma[x \mapsto b\ \mathit{list}] \vdash T : \tau$ for some τ, and $v_1, ..., v_n \in \mathcal{V}\llbracket \beta \rrbracket$. Then, for $x_1, ..., x_n$ fresh variables :*

$$\llbracket T \rrbracket^\bullet e[x \mapsto \langle v_1, ..., v_n \rangle] = \llbracket T[\texttt{[}x_1, ..., x_n\texttt{]}/x] \rrbracket^\bullet e[x_i \mapsto v_i]_i$$

5.4 Case Reduction

A basic valid program transformation in IMP is case reduction: a case instruction over a constant can be reduced to the appropriate branch.

We can directly prove that for a constant $\texttt{L} : \mathit{PROC}$ such that $\mathcal{I}(\texttt{L}) \equiv l\texttt{:(}\lambda\ \texttt{(}a_1\ ...\ a_n\texttt{)}\ c\texttt{)}$ from *Prg*, the following equivalence holds:

$$\begin{array}{l}\texttt{caseProc L of } l_1\texttt{:LAM(}\mathit{ids}, e\texttt{)=>}\ C_1 \\ \quad\texttt{|}\ l_2\texttt{:BIF =>}\ C_2 \\ \quad\texttt{|}\ l_3\texttt{:BPLUS =>}\ C_3 \\ \quad\texttt{| Stop =>}\ C_4\end{array} \qquad \equiv \quad C_1[l/l_1, \texttt{[}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{]}/\mathit{ids}, c/e]$$

Similar reductions can be obtained for the other possible values of `L`.

```
letrec                                       letrec
  fun f1(x1^β1_1,...,xn^β1_n)=C                fun f1(x1^β1_1,...,xn^β1_n)=C
  fun f2(y1^β2_1,...,ym^β2_m)=                 fun f2(y1^β2_1,...,ym^β2_m)=
        (C1;...;f1(V1,...,Vn);...)                   (C1;...;C[Vi/xi];...)
in D end                                     in D end
```

(a) IP_1 (b) IP_2

Figure 5.2: Call unfolding

We can also prove for a constant `C` : *CALL* such that $\mathcal{I}(\texttt{C}) \equiv l$:`(`$f\ a_1\ ...\ a_n$`)` from *Prg*, the following equivalence holds:

$$\begin{array}{l} \texttt{caseCall C of } l_1\texttt{:APP(}e, es\texttt{)=> } C_1 \\ \quad \texttt{| } l_2\texttt{:REC(}ids, es, e\texttt{)=>}C_2 \end{array} \quad \equiv \quad C_1[l_1/l, f/e, \texttt{[}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{]}/es]$$

In case `C` denotes a term c:`(letrec ((`$x_1\ d_1$`)...)` b`)`, the left term above is equivalent with $C_2[\texttt{[}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{]}/ids, \texttt{[}d_1\texttt{,}\ldots\texttt{,}d_n\texttt{]}/es, b/e, c/l_2]$.

Moreover, for a command `C` containing the free variable x such that $\Gamma[x \mapsto \mathit{PROC}] \vdash_C \texttt{C} : \mathit{unit}$, the term

$$\begin{array}{l} \texttt{caseX } x \texttt{ of} \\ \quad \texttt{l}_1 \texttt{ => C[l}_1/x] \\ \quad \vdots \\ \texttt{| l}_\Psi \texttt{ => C[l}_\Psi/x] \end{array}$$

is equivalent with `C`.

5.5 Call Unfolding

Another basic program transformation in IMP is that we can always inline the body of a function at the place where it is called, if the parameters are constants or variables.

Let IP_1 and IP_2 be the programs from Figure 5.2, considered as fully alpha-converted, i.e. no two variables have the same name. We also consider that $\texttt{V}_i \in \mathit{Val}$ such that $\Gamma \vdash_V \texttt{V}_i : \beta^1_i$, i.e., they are variables or constants. IP_2 is obtained from IP_1 by replacing the call to $\texttt{f}_1$ with its body and substituting the formal parameters with the actual ones. It is easy to see that if IP_1 typechecks, so does IP_2, under the same type.

Lemma 5.5.1. *IP_1 and IP_2 are equivalent.*

Proof. Let:

$$\begin{array}{rcl} F &=& \lambda(\phi_1,\phi_2).\lambda(a_1,...,a_n).\lambda\sigma.[\![\texttt{C}]\!]^C[\texttt{f}_1 \mapsto \phi_1, \texttt{f}_2 \mapsto \phi_2][\texttt{x}_i \mapsto a_i]_i\sigma \\ G &=& \lambda(\phi_1,\phi_2).\lambda(b_1,...,b_m).\lambda\sigma. \\ && \qquad [\![\texttt{(C}_1\texttt{;...;f}_1\texttt{(V}_1\texttt{,...,V}_n\texttt{);...)}]\!]^C[\texttt{f}_1 \mapsto \phi_1, \texttt{f}_2 \mapsto \phi_2][\texttt{y}_i \mapsto b_i]_i\sigma \\ G' &=& \lambda(\phi_1,\phi_2).\lambda(b_1,...,b_m).\lambda\sigma.[\![\texttt{(C}_1\texttt{;...;C[V}_i\texttt{/x}_i\texttt{];...)}]\!]^C[\texttt{f}_1 \mapsto \phi_1, \texttt{f}_2 \mapsto \phi_2][\texttt{y}_i \mapsto b_i]_i\sigma \end{array}$$

We define

$$\begin{array}{rcl} (f_1, f_2) &=& Fix(F,G) \\ (f'_1, f'_2) &=& Fix(F,G') \end{array}$$

To prove the lemma, suffices to prove that $(f_1, f_2) = (f'_1, f'_2)$.

From Bekić's theorem, the least fixed points can be computed as:

$$f_2 = Fix(\lambda g.G(Fix(\lambda f.F(f,g)),g))$$
$$f_2' = Fix(\lambda g.G'(Fix(\lambda f.F(f,g)),g))$$

To show that $f_2 = f_2'$, take an arbitrary g, and let $\theta = Fix(\lambda f.F(f,g))$. We need to show that $G(\theta, g) = G'(\theta, g)$. Because of the compositionality of our semantics, we just need to show that:

$$[\![\texttt{f}_1\texttt{(V}_1\texttt{,...,V}_n\texttt{)}]\!]^C e[\texttt{f}_i \mapsto \theta]\sigma = [\![\texttt{C[V}_i\texttt{/x}_i\texttt{]}]\!]^C e\sigma$$

It amounts to proving that:

$$\theta([\![\texttt{V}_1]\!]^V e, ..., [\![\texttt{V}_1]\!]^V e)\sigma = [\![\texttt{C[V}_i\texttt{/x}_i\texttt{]}]\!]^C e\sigma$$

Using the fact that $\theta = F(\theta, g)$ it amounts to proving that:

$$[\![\texttt{C}]\!](e[\texttt{x}_i \mapsto [\![\texttt{V}_i]\!]^V e]_i)\sigma = [\![\texttt{C[V}_i\texttt{/x}_i\texttt{]}]\!]^C e\sigma$$

which can be proven by a case analysis on each $\texttt{V}_i$, using Lemma 5.3.1 or Lemma 5.3.2

From this, we obtain that also $f_1 = f_1'$. □

The result of Lemma 5.5.1 can be easily extended to programs with multiple function definitions. It gives us a meaning-preserving program transformation: we can instantiate the body of a function at a place where that function is called, if the arguments passed are variables or constants of base type.

A similar argument can be used to prove that we can remove a function definition which is not used in the program.

Both transformations work in the reverse way too: we can add a new function to a IMP program. And, under certain conditions (given by Lemma 5.5.1), we can abstract a command into a function call.

5.6 Summary and Conclusions

We have proven several program transformation steps that capture the essence of partial evaluation: case reduction and call unfolding. The transformations yield more efficient programs. Indeed, by case reduction we eliminate a computation which might be performed many times in the program. By call unfolding we avoid the cost of a function call, possibly giving the compiler more optimization opportunities, at the expense of a larger code.

In the following developments, we use the program transformations in the specialization of *ICFA*. Further specialization opportunities, relying on properties of *ICFA* rather than being generally valid program transformations, will be encountered. We will account for them on a case by case basis.

Chapter 6

Specializing the Analysis

6.1 Introduction

We have defined in the previous chapter small transformation steps which yield equivalent terms: substitution, case reduction, call unfolding. The transformations can be used at any level of the program, where they are applicable, due to the compositionality of the semantics.

In the following, we describe what transformations we apply to *ICFA* to achieve specialization. Although the specialization process is specific to the program *ICFA*, we present a concise and convincing way to prove the correctness of specialization.

6.2 Polyvariant Specialization

Our observation is that considering the first parameter `exp` of the `Fhat` function to be static enables a significant amount of static reductions. It is a typical case of bounded static variation [9], as `exp` is ranging over a finite domain. By introducing a case over its values at the entry point of the `Fhat` function we can consider `exp` as static inside each branch of the case.

This can be done directly in our case. Using the equivalence proved in Section 5.4, in the definition of `Fhat` we can replace its body `FH` as follows:

```
fun Fhat(exp,Vals) =  caseX exp of
      $l_1$ => FH[$l_1$/exp]
      ⋮
    | $l_\Psi$ => FH[$l_\Psi$/exp]
```

and obtain an equivalent program. By the reverse of Lemma 5.5.1 and its derivatives, the above definition can be replaced with the following series of definitions:

```
fun Fhat$_{l_1}$(Vals) = FH[$l_1$/exp]
        ⋮
fun Fhat$_{l_\Psi}$(Vals) = FH[$l_\Psi$/exp]
fun Fhat(exp,Vals)=
  caseX exp of
      $l_1$ => Fhat$_{l_1}$(Vals)
      ⋮
    | $l_\Psi$ => Fhat$_{l_\Psi}$(Vals)
```

6.3 Further Transformation

We examine the body of one of the $\mathtt{Fhat}_{\mathtt{l}_i}$ functions, namely $\mathtt{FH}[\mathtt{l}_i/\mathtt{exp}]$. It expands to: $\mathtt{caseProc}\ \mathtt{l}_i\ \mathtt{of}\ \ldots.$ Using the equivalences in Section 5.4 we can reduce the case on the constant $\mathtt{l}_i$.

In the case when $\mathtt{l}_i$ denotes a term l:$(\lambda\ (a_1\ ...\ a_n)\ e)$ from *Prg*, by case reduction we obtain:

$$\mathtt{(UpdateNRho([}a_1\mathtt{,...,}a_n\mathtt{],Vals);Chat(}e\mathtt{))}$$

Since e is a constant, by Lemma 5.5.1, the above is equivalent to:

$$\mathtt{(UpdateNRho([}a_1\mathtt{,...,}a_n\mathtt{],Vals);\ caseCall}\ e\ \mathtt{of\ ...)}$$

Again, the `caseCall` can be reduced, which produces either a call to `Set.Fmap` or a call to `Chat`. In the later case, we can continue to unfold the call to `Chat` and reduce the `caseCall` as long as possible. The process is determined by the term $\mathtt{l}_i$ and it is provably finite.

We perform the above transformations to each function $\mathtt{Fhat}_{\mathtt{l}_i}$. At the end, none of the $\mathtt{Fhat}_{\mathtt{l}_i}$ functions are referring to `Chat` anymore: all calls to it have been unfolded. We can remove its definition.

We observe that in all its occurrences, `AhatN` is now called with a constant parameter. For each application c:$(f\ a_1\ ...\ a_n)$ in *Prg* we obtain a call $\mathtt{AhatN([}a_1\mathtt{,...,}a_n\mathtt{])}$. We can use the equivalence:

$$\mathtt{AhatN([}a_1\mathtt{,...,}a_n\mathtt{])} \equiv \mathtt{[Ahat(}a_1\mathtt{),...,Ahat(}a_n\mathtt{)]}$$

to eliminate the traversal of a list. Also, from a definition c:$\mathtt{(letrec\ ((}x_1\ d_1\mathtt{)}...\mathtt{(}x_n\ d_n\mathtt{))}\ e\mathtt{)}$ in the target program *Prg* we obtain

$$\mathtt{UpdateNRho([}x_1\mathtt{,...,}x_n\mathtt{],AhatN([}d_1\mathtt{,...,}d_n\mathtt{]))}$$

in the residual program. The above term is provably equivalent to:

$$\mathtt{UpdateRho(x_1,Ahat(d_1));...;UpdateRho(x_}n\mathtt{,Ahat(d_}n\mathtt{))}$$

and we remove again the overhead incurred by constructing/destructing a list.

At this point, each occurrence of `Ahat` is a call with a constant parameter. Since, in fact, `Ahat` involves a case over the syntax of its parameter, we can replace it by the following equivalences:

$$\begin{aligned}
\mathtt{Ahat(}l\mathtt{:(}\lambda\ \mathtt{(}id_1\ ...\ id_n\mathtt{)}\ e\mathtt{))} &\equiv \mathtt{Set.Single(}l\mathtt{)}\\
\mathtt{Ahat(}x\mathtt{)} &\equiv \mathtt{lookup(}x\mathtt{)}\\
\mathtt{Ahat(}c\mathtt{)} &\equiv \mathtt{Set.Empty}\\
\mathtt{Ahat(}p\mathtt{:if)} &\equiv \mathtt{Set.Single(}p\mathtt{)}\\
\mathtt{Ahat(}p\mathtt{:+)} &\equiv \mathtt{Set.Single(}p\mathtt{)}
\end{aligned}$$

The process is finite, and by a counting argument, we can prove that the resulting program is linear in the size of the target program *Prg*. Moreover, the resulting program has no more case instructions on the syntax of *Prg*.

An example illustrating all the mentioned transformations for a given procedure $\mathtt{l}_i$ can be seen in Figure 6.1.

We consider $ICFA'$ as in Figure 6.2 to be the resulting program.

6.4 Arity Raising and Dispatching

The residual program $ICFA'$ is still not satisfactory. Let $\alpha_1, ..., \alpha_\Psi$ be the arities of the procedures $\mathtt{l}_1, ..., \mathtt{l}_\Psi$ from the source program *Prg*, and let Θ be the greatest of them, i.e., the maximum arity encountered in *Prg*. We can still identify cases where the computation is strictly dependent on the syntax of *Prg*:

```
                    l:(λ (n C) r:(letrec ((loop l_k:λ)) c:(loop n C)))

fun Fhat_{l_i}(Vals)=  (UpdateNRho([n,C],Vals);
                        UpdateRho(Loop,Set.Single(l_k));
                        if Older(c) then
                           (Mark(c);
                            UpdateDelta(c,lookup(loop));
                            Set.Fmap(Fhat,lookup(loop),[lookup(n),lookup(C)]))
                        else skip)
```

Figure 6.1: Example of residual function

```
letrec
  fun Fhat_{l_1}(Vals) = C_1
      ⋮
  fun Fhat_{l_Ψ}(Vals) = C_Ψ
  fun Fhat(exp,Vals) =
    caseX exp of
      ⋮
    | l_i => Fhat_{l_i}(Vals)
      ⋮
in Fhat(P,Start) end
```

Figure 6.2: The $ICFA'$ program

- We can see that all the occurrences of `Set.Fmap` inside the new program are in one of the two forms:

$$\texttt{Set.Fmap(Fhat,Ahat(f),[Ahat(a}_1\texttt{),...,Ahat(a}_k\texttt{)])}$$
$$\texttt{Set.Fmap(Fhat,List.nth(Vals,1),nil)}$$

- When calling `Set.Fmap`, we create a list, which length is determined by the syntax of *Prg*. Inside `Set.Fmap` we need to recompute the length of the list. The list is later traversed and destroyed.

- The list `Vals` received by $\texttt{Fhat}_{\texttt{l}_i}$ is expected to have a fixed length, namely α_i. It is more efficient to pass each element of the list as a separate parameter rather than constructing and destructing a list.

In the same time, for efficiency, we want to implement the `caseX` as a table lookup. We take the specialization further to make account of the above observations.

An easy solution to avoid constructing the lists would be to add to our language several versions of `Set.Fmap`, "specialized" with respect to the length of the list received as the third parameter. We could replace all its occurrences with the specialized versions and obtain:

$$\texttt{Set.Fmap}_k\texttt{(Fhat,Ahat(f),Ahat(a}_1\texttt{),...,Ahat(a}_k\texttt{))}$$
$$\texttt{Set.Fmap}_0\texttt{(Fhat,List.nth(Vals,1))}$$

However, the specialized built-ins $\texttt{Set.Fmap}_i$ still need to construct the same list in order to be passed to the `Fhat` function. To completely remove the overhead of the lists, we also to specialize $\texttt{Fhat}_{\texttt{l}_i}$ by arity raising with respect to the expected length of the list received as parameter.

On the other side, to eliminate the `caseX` construct, we want to have a built-in facility of indexing the top-level function definitions. When given a number i we want to call the i-th

```
letrec
  fun Fhat$_{1_1}$(x$_1$,...,x$_{\alpha_1}$)=C$'_1$[[x$_1$,...,x$_{\alpha_1}$]/Vals]
      ⋮
  fun Fhat$_{1_\Psi}$(x$_1$,...,x$_{\alpha_\Psi}$)=C$'_\Psi$[[x$_1$,...,x$_{\alpha_\Psi}$]/Vals]
in Fhat$_{Prg}$(Start) end
```

Figure 6.3: $ICFA''$ program

```
                    l:(λ (n C) r:(letrec ((loop l_k:λ)) c:(loop n C)))

fun Fhat$_{1_i}$(x$_1$,x$_2$) =  (UpdateNRho([n,C],[x$_1$,x$_2$]);
                         UpdateRho(Loop,Set.Single(l$_k$));
                         if Older(c) then
                            (Mark(c);
                             UpdateDelta(c,lookup(loop));
                             dispatch$_2$(lookup(loop),lookup(n),lookup(C)))
                         else skip)
```

Figure 6.4: Example of residual function, after arity raising

defined function. Providing that facility raises another problem: the indexing function would have a dependent type, which is not part of IMP.

A complete solution, which accounts both for the arity raising and for eliminating the `caseX` construct, without extending the IMP language with other constructs, can be done by just altering the semantics of the `letrec` construct of IMP. Our solution is to hide the dependent type inside predefined constants of the language. We provide at semantic level several dispatch functions replacing the `Set.Fmap`$_j$ builtin, for each $0 \leq j \leq \Theta$.

The intended rezidualized program $ICFA''$ is to be as shown in Figure 6.3. It is obtained from $ICFA'$ by removing the definition of `Fhat`. Each command $\mathtt{C}'_i$ is obtained from $\mathtt{C}_i$ by replacing all occurrences of `Set.Fmap(Fhat,`S`,[`A_1`,...,`A_k`])` with `dispatch`$_k$`(`S`,`A_1`,...,`A_k`)`, for any terms S and A_i and for every $k \geq 0$. An example of how the function in Figure 6.1 is modified can be found in Figure 6.4.

Informally, the `dispatch`$_i$ behave identical to `Set.Fmap`. They however, do not receive the `Fmap` function as a parameter, they directly call `Fhat`$_{1_i}$, which are extracted from the `letrec` construct.

For a formal account of the above transformation, we have restated the semantics of the `letrec` construct of IMP to define `dispatch`$_i$ mutually recursive with the top-level functions. We have been able to prove that with this construction, the result computed by $ICFA''$ is equal with the one given by $ICFA'$. For space reasons, we do not present the construction here.

The final program transformation we perform is to eliminate all list manipulation. Now the only place where lists appear is only as:

`UpdateNRho([`id_1`,...,`id_n`],[V`$_1$`,...,V`$_n$`])`

inside a `Fhat`$_k$ function, where $\mathtt{L}_k = l$`:(`λ `(`id_1 ... id_n`)` e`)`. This call is also equivalent to:

`UpdateRho(id`$_1$`,V`$_1$`);...;UpdateRho(id`$_n$`,V`$_n$`)`

6.5 Example

In Figure 6.6 we present a the specialized version of $ICFA$, which is obtained for the CPS-version of a simple program given in Figure 6.5. The specialized version corresponds with the output of our specializer, and we can see the implementation of `dispatch`$_1$ function.

6.6 Summary and Conclusions

We have presented a method for specializing *ICFA* by means of meaning-preserving program transformations. By the described specialization, we have removed all the case instruction on the syntax of the target program *Prg*, and all operations with lists. All the computation performed by the residual program is operations with the variable environment and call cache.

In our implementation, the specialized program is produced by a “generating extension” [9].

From a practical point of view, the language IMP is a subset of ML. Thus, all we need to do to run a IMP program in an ML environment is to implement the predefined constants. To perform the dispatching over the function declarations, we need to define a mutable array of functions, which will be initialized to the top-level functions, before the evaluation of the body of `letrec`. We can implement the `dispatch`$_i$ functions to make use of the array in an obvious manner, as it can be seen in Figure 6.6.

In the following chapter, we present the experimental results concerning the efficiency of the specialized analysis with respect to the standard implementation.

```
((lambda (x) x)
      (lambda (y) y))
```

(a) Direct-style input program

$(\text{lambda}^{2}\ (\text{C1}^{1})$
$((\text{lambda}^{6}\ (\text{x}^{4}\ \text{C2}^{5})\ (\text{C2}^{8}\ \text{x}^{9})^{7})$
$(\text{lambda}^{12}\ (\text{y}^{10}\ \text{C3}^{11})\ (\text{C3}^{14}\ \text{y}^{15})^{13})$
$(\text{lambda}^{17}\ (\text{V1}^{16})\ (\text{C1}^{19}\ \text{V1}^{20})^{18})^{3}))$

(b) CPS, labeled version

Figure 6.5: Sample Program

```
datatype Fun = FUN1 of ((label Set.Set) -> unit)
             | FUN2 of ((label Set.Set * label Set.Set) -> unit)
fun dispatch1(S,S1) = let fun loop [] = ()
                            | loop (l::ls) = ((case Array.sub(FTable,l) of
                                     FUN1(f)=> f(S1)
                                   | _ => ());loop ls)
                      in loop S end
fun Fhat1(X1,X2) = (UpdateRho(4,X1);UpdateRho(5,X2);
                    if Older(7) then
                       let val S1 = lookup(5)
                       in (UpdateDelta(7,S1); dispatch1(S1,lookup(4))) end
                    else ())
fun Fhat2(X3,X4) = (UpdateRho(10,X3);UpdateRho(11,X4);
                    if Older(13) then
                       let val S3 = lookup(11)
                       in (UpdateDelta(13,S3); dispatch1(S3,lookup(10))) end
                    else ())
fun Fhat3(X5) = (UpdateRho(16,X5);
                 if Older(18) then
                    let val S4 = lookup(1)
                    in (UpdateDelta(18,S4); dispatch1(S4,lookup(16))) end
                 else ())
val _ = (Array.update(FTable,17,FUN1(Fhat3));
         Array.update(FTable,12,FUN2(Fhat2));
         Array.update(FTable,6,FUN2(Fhat1)))
fun start(X6) = (UpdateRho(1,X6);
                 if Older(3) then
                     let val S2 = Set.Single(6)
                     in (UpdateDelta(3,S2);
                         dispatch2(S2,Set.Single(12),Set.Single(17)))
                     end
                 else ())
```

Figure 6.6: Example of residual program

Chapter 7

Benchmarks

7.1 Introduction

In order to assess the efficiency of the partial evaluation we have compared the running times of the standard analyzer, as given in Figure 4.8, and of the specialized analyzer.

Our implementation language is SML/NJ [1]. The translation to ML of the program *ICFA* serves as our implementation of the standard analyzer. The specialized analyzer is obtained by a "generating extension" [9], which directly produces the ML translation of the specialized program.

We have implemented a front-end which parses a Scheme program, alpha-converts it and translates it to a reduced subset of Scheme. The result is then translated to CPS and passed either to the standard analyzer or to the generating extension. In the later case, a residual program is produced, which is the loaded and run.

The front-end of our benchmarks has also been used in the ongoing work mentioned in Section 1.1 to provide source programs for experiments with partial evaluation of 0-CFA in constraint form.

7.2 Results

We have considered Feeley's Scheme benchmarks:

conform	A program that manipulates lattices and partial orders
earley	A parser generator for context-free languages based on Earley's algorithm.
interp	An environment-based call-by-need λ-calculus interpreter
lambda	A substitution-based call-by-name λ-calculus interpreter
lex	A generator of lexical analyzers
ll1	An $LL(1)$ parser generator
peval	A small partial evaluator for Scheme
source	A parser for Scheme

We measure the relative speedup gained by specialization. We only count the time it takes to perform the analysis, both in the standard form and specialized form, not including the time spent to generate the residual programs.

In Table 7.1 we show the relative speedups obtained running the benchmarks over several platforms: an Ultra-4 Sparc with 1Gb RAM, an SGI O2 with 128Mb RAM and a Linux PC with 64Mb RAM. The table gives in the second column the number of lines of the benchmark, in the third the number of functions generated in the residual program and in the fourth column the average size of a set in the variable environment. They consistently show that specializing the analyzer yields a speedup of between 30% and 55%.

Program	Size (lines)	Residual functions	Average set size	Sun	SGI	Linux
conform	601	814	0.38	2.03	1.87	1.85
earley	657	776	0.49	2.03	1.86	1.83
interp	479	549	0.94	2.09	1.61	1.71
lambda	690	1176	0.40	1.77	1.62	1.73
lex	1253	1586	0.35	1.94	1.64	1.89
ll1	652	783	0.37	2.11	2.07	1.93
peval	690	1238	0.91	1.93	1.46	1.68
source	476	376	0.36	2.32	2.37	2.20

Table 7.1: Benchmarks results

7.3 Summary and Conclusions

The results obtained in the benchmarks are encouraging. They show that partial evaluation pays off in the case of the 0-CFA.

Several questions remain though unanswered. Since the analysis is run only once, for the method to be successful, it is necessary that the time spent to generate and compile the residual program is substantially less that the analysis itself.

One answer can be found in the language IMP. It can be seen that the language is a very low-level one, and its translation in a machine-like language is just incurring a constant factor.

The other answer relies on the complexity estimates of the analysis. From a complexity point of view, the analysis we specialize is a cubic-time worst-case algorithm with a worst-case quadratic space demand. We reduce the running time by a constant factor, introducing an additional linear-time computation and a linear space demand.

Since the size of the residual program is linear and the time spent to produce it is also linear, the code generation process with eventually be faster that the cubic worst-case complexity of the analysis.

We have also experimented with several ideas on improving the code generation. For instance, the first-order call graph (which can be precomputed in linear time), can enable us to avoid generating code for parts of the program that are not analyzed. Moreover, we can precompute the first order control flow information and avoid generating code which will compute it. Preliminary results show that a further speedup of 30% can be obtained.

Chapter 8

Conclusion and Perspectives

We have presented in this report a case-study of applying partial evaluation to program analysis. Starting from Shivers's CFA over CPS terms of a higher-order language, we have obtained an analyzer specialized with respect to the input program. We have presented the results obtained when comparing the specialized analysis to the standard one over a set of benchmarks.

In this work, we identify two main contributions. First, we have established a formal connection between the specification of Shivers's CFA and its proposed implementation. Secondly, using an uniform programming-language approach, we have established correctness of the implementation of the analysis and we have obtained certified specialized analyzers. Also, we have implemented the analysis and its specialization, and produced experimental evidence of the efficiency of our results.

We consider the novelty our approach as presenting a unified view towards the formalization of a problem usually treated in an informal fashion. Reasoning in the world of programming languages and formal semantics enables us to provide a clear and simple certification of the practical idea of using partial evaluation in program analysis.

In this sense, our work comes to justify and formalize part of the ideas of Boucher and Feeley's work [3]. We have not considered though whether our approach could be extended to their broader concerns, like abstract compilation [2].

The work we have presented we regard as a complete study, as we have managed to complete the road from the theoretical statement of a problem to the practical implementation and experimental evaluation of the results. However, an unanswered question is whether the developments in this work could be generalized.

One interesting issue would be whether the result obtained in Chapter 3 could be transformed into a framework in which a monotone recursive function could be transformed into a version with single-threaded arguments, which is still "safe" with respect to a defined criteria.

On the partial evaluation side, the specialization based on the small-steps reductions in our approach is in contrast with the main trend of automated specialization based on program annotation. However, it would be useful to know how and what are the frameworks in which non-formalized transformations — such as polyvariant specialization or arity raising, which we perform here on a particular case — could be abstracted in generic partial evaluation frameworks, as, for instance, type directed partial evaluation.

Bibliography

[1] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Jan Maluszyński and Martin Wirsing, editors, *Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 1–13, Passau, Germany, August 1991. Springer-Verlag.

[2] Dominique Boucher and Marc Feeley. Un système pour l'optimisation globale de programmes d'ordre supérieur par compilation abstraite séparée. Technical Report 992, Université de Montréal, 1995.

[3] Dominique Boucher and Marc Feeley. Abstract compilation: a new implementation paradigm for static analysis. In *1996 International Conference on Compiler Construction*, 1996.

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

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

[6] Marc Feeley. Deux approches à l'implantation du language Scheme. Master's thesis, Université de Montréal, May 1986.

[7] Marc Feeley and Guy Lapalme. Using cloures for code generation. *Journal of Computer Languages*, 12(1):47–66, 1987.

[8] Fritz Henglein. Simple closure analysis. Technical Report Semantics Report D-193, DIKU, Computer Science Department, University of Copenhagen, 1992.

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

[10] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[11] Jens Palsberg and Michael I. Schwartzbach. Object-Oriented Type Inference. In *Proceedings of the OOPSLA '91 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 146–161, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.

[12] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference for partial types. *Information Processing Letters*, 43(4):175–180, September 1992.

[13] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972.

[14] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[15] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.

[16] Glynn Winskel. *The Formal Semantics of Programming Languages.* Foundation of Computing Series. The MIT Press, 1993.