KARLSRUHER INSTITUT FÜR TECHNOLOGIE
FAKULTÄT FÜR INFORMATIK

Joachim Breitner

Student Research Project

Control Flow in Functional Languages

Formally taming lambdas

Supervisors:
Prof. Dr. Gregor Snelting
Andreas Lochbihler

November 15, 2010

Abstract

In his dissertation[9], Olin Shivers introduces a concept of control flow graphs for functional languages, provides an algorithm to statically derive a safe approximation of the control flow graph and proves this algorithm correct. In this student research project, Shivers' algorithms and proofs are formalized using the theorem prover system Isabelle.

Contents

CHAPTER 1

Introduction

CONTROL Flow Analysis plays an important role in compiler construction as well as in security analysis. Having the latter case in mind, Daniel Wasserrab provides a formally verified slicing framework acting on an abstract and language-independent control flow graph in his dissertation[11]. This framework has been instantiated for programs written in either a simple toy imperative language or the Java-like language Jinja[6].

These programming languages are imperative. Therefore, we are interested in its applicability to functional languages. This requires a notion of a control flow graph for functional programs. In 1991, Olin Shivers defined such a control flow graph and developed a static analysis algorithm to calculate it. Later research on functional control flow is often based on his definition.

To be able to connect to Wasserrab's framework, the definitions and algorithms of Shivers need to be formalized in the theorem prover system Isabelle. This is the main goal of this project, although we treat the actual connection only theoretically.

This document starts with a concise overview of Shivers' approach in Chapter 2. Then we explain the prototype in Haskell (Chapter 3) as well as the formalization in Isabelle (Chapter 4), stating where it differs from the original. We motivate some of the more interesting choices in the formalization, such as the use of *HOLCF*. Some lemmas took several failed or unsatisfying attempts, which we do not hush up. We took special care to make the appearance of the documents generated by Isabelle as similar to the original as possible. Section 4.6 presents the tricks used to that end. Chapter 5 explains how these results could be connected to Wasserrab's framework.

A total of 3632 lines of Isabelle code (including comments) and 890 lines of Haskell code (including 199 lines of commentary) were written.

CHAPTER 2

Taming Lambdas

FUNCTIONAL languages, i.e. programming languages that treat computations as first class citizens, are harder to tackle by control flow analysis than imperative languages. For the latter, a function call names the function that will be called and a static analysis can easily trace the control flow at that point. In functional languages, the callee can be a variable which, in turn, can stand for any computation that was stored somewhere else in the program.

Additionally, such a stored computation is a *closure*, i.e. it remembers the variable assignments from the time the closure was created. Therefore, a variable can have more than one current value at any given point in the program execution, where different closures see different values. This is an additional issue when trying to statically make statements about a functional program's control flow.

Shivers approaches this problem in his 1991 Ph.D. thesis, subtitled "taming lambdas". He defines an exemplary functional language in *continuation-passing style* (CPS). CPS means that the return value of functions is not actually returned, but rather passed on to a continuation function, which is provided as an argument. Consider for example the following code :

$\mathsf{main} = \textbf{print}\ ((\mathsf{x} + \mathsf{y}) * (\mathsf{z} - \mathsf{w}))$

In continuation-passing style, the each of the operations $+$, $*$, $-$ and **print** takes a continuation argument, and the main function is being passed a top-level continuation:

$\mathsf{main\ c} = +\ \mathsf{x\ y}\ (\lambda \mathsf{xy}.\ -\ \mathsf{z\ w}\ (\lambda \mathsf{zw}.\ *\ \mathsf{xy\ zw}\ (\lambda \mathsf{prod}.\ \textbf{print}\ \mathsf{prod\ c})))$

$$
\begin{array}{rll}
\text{PR} & ::= & \text{LAM} \\
\text{LAM} & ::= & (\lambda\ (v_1 \ldots v_n)\ c) \qquad [v_i \in \text{VAR},\ c \in \text{CALL}] \\
\text{CALL} & ::= & (f\ a_1 \ldots a_n) \qquad [f \in \text{FUN},\ a_i \in \text{ARG}] \\
 & & (\texttt{letrec}\ ((f_1\ l_1) \ldots)\ c) \qquad [f_i \in \text{VAR},\ l_i \in \text{LAM},\ c \in \text{CALL}] \\
\text{FUN} & ::= & \text{LAM} + \text{REF} + \text{PRIM} \\
\text{ARG} & ::= & \text{LAM} + \text{REF} + \text{CONST} \\
\text{REF} & ::= & \text{VAR} \\
\text{VAR} & ::= & \{\texttt{x},\texttt{y},\texttt{foo},\ldots\} \\
\text{CONST} & ::= & \{\texttt{3},\texttt{\#f},\ldots\} \\
\text{PRIM} & ::= & \{\texttt{+},\texttt{if},\texttt{test-integer},\ldots\} \\
\text{LAB} & ::= & \{l_i, r_i, c_i, \ldots\}
\end{array}
$$

Figure 1: CPS syntax

2.1 Syntax

The syntax of Shivers' toy functional programming language is given in Figure 1. A program (PR) is a lambda expression. A lambda expression (LAM) abstracts a call. Calls (CALL) either call a function with the given argument list or bind lambdas to names. Such calls are named letrec because the bound lambdas may refer to each other in a possibly mutually recursive fashion, making the language Turing-complete.

Function values (FUN) can be lambda expressions, references to variables or primitive operations, while in argument positions (ARG), primitive operations are disallowed and constant expressions are allowed. References to variables (REF) name the referenced variable. There is a set of variable names (VAR). Constants (CONST) are either integers or #f for a false value. A number of primitive operations (PRIM) are defined. Shivers treats mutable references and adds corresponding primitive operations in later chapters of his dissertation. We omit this and concentrate on the primitive operations `+` and `if`.

Although not given explicitly, every lambda, call, constant, variable reference and primitive operation is tagged with a unique label from the set LAB. These can be thought of as positions in the program source – for our purpose they

$$
\begin{aligned}
\text{Bas} &= \mathbb{Z} + \{\text{false}\} & \mathcal{PR} &: \text{PR} \rightharpoonup \text{Ans} \\
\text{Clo} &= \text{LAM} \times \text{BEnv} & \mathcal{A} &: \text{ARG} \cup \text{FUN} \to \text{BEnv} \to \text{VEnv} \to \text{D} \\
\text{Proc} &= \text{Clo} + \text{PRIM} + \{\mathit{stop}\} & \mathcal{C} &: \text{CALL} \to \text{BEnv} \to \text{VEnv} \to \text{CN} \rightharpoonup \text{Ans} \\
\text{D} &= \text{Bas} + \text{Proc} & \mathcal{F} &: \text{Proc} \to \text{D}^* \to \text{VEnv} \to \text{CN} \rightharpoonup \text{Ans} \\
\text{CN} &= \mathit{(contours)} \\
\text{BEnv} &= \text{LAB} \rightharpoonup \text{CN} \\
\text{VEnv} &= (\text{VAR} \times \text{CN}) \rightharpoonup \text{D} \\
\text{Ans} &= (\text{D} + \{\mathit{error}\})_\perp
\end{aligned}
$$

Figure 2: CPS semantics domains

are just an abstract set. Additional *internal labels* are added to this set for primitive operations, representing the internal call sites. For example the `if` primitive operation is being passed two continuation that might be called, one for the true case and one for the false case. Therefore, two internal labels are associated with this primitive operation. Also we assume that programs are alphatised, i.e. each variable name v is bound at exactly one position, whose label is given by *binder* v.

2.2 Standard Semantics

Shivers gives a denotational semantics for the above language, that is a partial function $\mathcal{PR}$ from the set of programs to the set of integers with an additional element indicating run-time errors (Ans). Following the structure of the syntax tree, he defines functions $\mathcal{C}$, $\mathcal{F}$ and $\mathcal{A}$ that evaluate call expressions, apply arguments to procedures and evaluate argument expressions, respectively. Their domains and ranges are given in Figure 2. The rather lengthy equations of their definitions are omitted here.

Semantic values (D) can either be basic values (Bas) which consist of the integers plus the value representing false, or a procedure. Procedures (Proc) again are either a closure (Clo), which is a lambda expression bundled with a context, a primitive operation or *stop*.

The special value *stop* is the continuation initially passed to the program. When this is eventually called, either in a CALL expression or as the con-

tinuation of a primitive operation, the evaluation comes to a halt and the argument to *stop* is the result of the evaluation. If a run-time error occurs (wrong number of arguments, callee not a procedure, undefined variable lookup), *error* is returned.

The semantics functions are partial functions, indicated by the harpoon instead of the arrow and by the $\perp$ annotation of the set Ans of answer values. Functional programs are Turing-complete and therefore possibly nonterminating. For such programs, the semantics is defined to return $\perp$.

The evaluation context needs to be passed along the semantics functions and closure values need to encapsulate the context at the time the lambda expression is evaluated to a closure. The context information is separated into two maps here: The global *variable environment* (VEnv) and the lexical *contour environment* (or binding environment, BEnv, in the following usually denoted by β). The variable environment can store multiple values for each variable name. These are differentiated by a *contour number* from the set CN. These can be thought of time stamps which are increased in every evaluation step. When a variable is bound to a new value, it is stored in the variable environment with the current contour number. The contour environment tells for each binding position (lambda or `letrec` expression) which contour counter is to be used when looking up a variable bound there. By storing the contour environment within a closure and using it when evaluating the call inside the closure, the correct value for each variable binding is accessed.

The set CN of contours is not given explicitly here. Instead, we will treat it abstractly and only state the properties we expect from this set: There needs to be an initial contour b_0 and a function $nb\colon \mathrm{CN} \to \mathrm{CN}$ which generates new contours. The set of contours is partially ordered and if b is greater or equal than all allocated contours, then $nb\ b$ is strictly greater than all allocated contours. It is easy to see that the natural numbers with $b_0 = 0$ and $nb\ b = \mathrm{Suc}\ b$ fulfill the requirements, but in the later proofs it will be convenient to choose other sets carrying more information.

2.3 Exact nonstandard semantics

At the moment we are not so much interested in the value a program returns but rather the calls that occur while evaluating the program. To that end we alter the standard semantics introduced in the previous section to calculate the *call cache*. This is a partial map from call-site/contour-environment pairs

$$\mathcal{C}\ (c : (f\ a_1 \ldots a_n))\ \beta\ ve\ b = \begin{cases} [], & f' \notin \text{Proc} \\ (\mathcal{F}\ f'\ av\ ve\ b')[(c, \beta) \mapsto f'], & \text{otherwise} \end{cases}$$
$$\text{where} \quad f' = \mathcal{A}\ f\ \beta\ ve$$
$$av_i = \mathcal{A}\ a_i\ \beta\ ve$$
$$b' = nb\ b$$

Figure 3: The exact nonstandard semantics for a call expression

to procedures. We therefore modify the domains of the semantics in Figure 2 on page 11 as follows:

$$\text{CCache} = (\text{LAB} \times \text{BEnv}) \rightharpoonup \text{Proc}$$
$$\text{Ans} = \text{CCache}$$

The semantics function definitions are extended to record the calls as they happen. Figure 3 shows exemplary the equation of $\mathcal{C}$ for a call expression $(f\ a_1 \ldots a_n)$ with label c. If the syntactic value f is evaluated to a procedure $f' \in \text{Proc}$, evaluation continues by applying $\mathcal{F}$ to f' and the arguments. This returns a call cache, which is amended by the entry $(c, \beta) \mapsto f'$ to reflect the current call before the updated call cache is returned by C.

This semantics evaluates the program in full, just like the standard semantics, and is therefore unsuitable to be used in a static analysis. Nevertheless it is the theoretic ideal that a statically calculated call cache will be compared against.

2.4 Abstract nonstandard semantics

Shivers then proceeds to define a nonstandard semantics that can be calculated statically. The trade off is that the call cache obtained now is just an approximation to the exact call cache, i.e. any observed call in the exact call cache will be in the abstract call cache, but the converse does not need to hold.

Figure 4 on the following page gives the domains of the abstract semantics functions. The abstract semantics $\widehat{\mathcal{PR}}$ will run the program no longer in full detail. It will not keep track of actual values computed and conditional

$$\widehat{\text{Clo}} = \text{LAM} \times \widehat{\text{BEnv}} \qquad \widehat{\text{BEnv}} = \text{LAB} \rightharpoonup \widehat{\text{CN}}$$
$$\widehat{\text{Proc}} = \widehat{\text{Clo}} + \text{PRIM} + \{\mathit{stop}\} \qquad \widehat{\text{VEnv}} = (\text{VAR} \times \widehat{\text{CN}}) \rightharpoonup \widehat{\text{D}}$$
$$\widehat{\text{D}} = \mathcal{P}(\widehat{\text{Proc}}) \qquad \widehat{\text{CCache}} = (\text{LAB} \times \widehat{\text{BEnv}}) \to \widehat{\text{D}}$$
$$\widehat{\text{CN}} = \mathit{(contours, finite)} \qquad \widehat{\text{Ans}} = \widehat{\text{CCache}}$$

$$\widehat{\mathcal{PR}} : \text{PR} \rightharpoonup \widehat{\text{Ans}}$$
$$\widehat{\mathcal{A}} : \text{ARG} \cup \text{FUN} \to \widehat{\text{BEnv}} \to \widehat{\text{VEnv}} \to \widehat{\text{D}}$$
$$\widehat{\mathcal{C}} : \text{CALL} \to \widehat{\text{BEnv}} \to \widehat{\text{VEnv}} \to \widehat{\text{CN}} \rightharpoonup \widehat{\text{Ans}}$$
$$\widehat{\mathcal{F}} : \widehat{\text{Proc}} \to \widehat{\text{D}}^* \to \widehat{\text{VEnv}} \to \widehat{\text{CN}} \rightharpoonup \widehat{\text{Ans}}$$

Figure 4: Abstract nonstandard semantics domains

branches taken. Instead, it assumes that every branch of an `if` statement is possibly taken. Semantic values are now sets of possible procedures. This can be seen in Figure 5, which contains both the exact and the abstract nonstandard semantics definitions for a conditional expression. The continuations respectively, in the abstract case, the sets of possible continuations are passed as the arguments c_0 and c_1.

After removing the plain values from our semantics domains, the remaining sets would all be finite – if $\widehat{\text{CN}}$ was finite. This is the main trick to obtain a computable abstract semantics. If $\widehat{\text{CN}}$ is finite, there are only finitely many possible closures, procedures, binding environment, variable environments and call caches, and our function becomes computable.

Naturally, there is a loss of information when replacing the infinite set of contour counters in the exact case by a finite set. Variable bindings that could otherwise be distinguished by their contour counter now fall together. Therefore, the abstract variable environment tracks sets of possible values for each variable/contour counter pair.

Again, the choice of the set $\widehat{\text{CN}}$ is not fixed. Possible choices include the singleton set, resulting in "0th-order Control Flow Analysis" (0CFA) or the set of lambdas representing the call site a lambda was called from, resulting in first-order Control Flow Analysis (1CFA). To be able to track this information, the function $\widehat{nb}$, used to generate new contour counters, takes such a label as

$$\mathcal{F}\ \texttt{if}\ [v, c_0, c_1]\ ve\ b = \begin{cases} [], \text{ if } c_0 \notin \text{Proc or } c_1 \notin \text{Proc} \\ \{(ip_0, \beta) \mapsto c_0\} \cup \mathcal{F}\ c_0\ []\ ve\ (nb\ b), \text{ if } v \neq 0 \\ \{(ip_1, \beta) \mapsto c_1\} \cup \mathcal{F}\ c_1\ []\ ve\ (nb\ b), \text{ if } v = 0 \end{cases}$$
$$\text{where } \beta = \{p \mapsto b\}$$

$$\begin{aligned} \widehat{\mathcal{F}}\ \texttt{if}\ [v, c_0, c_1]\ ve\ b = & \bigcup_{f \in c_0} \widehat{\mathcal{F}}\ f\ []\ ve\ (\widehat{nb}\ b\ ip_0) \\ & \cup \bigcup_{f \in c_0} \widehat{\mathcal{F}}\ f\ []\ ve\ (\widehat{nb}\ b\ ip_1) \\ & \cup \{(ip_0, \beta) \mapsto c_0,\ (ip_1, \beta) \mapsto c_1\} \\ & \text{where } \beta = \{p \mapsto b\} \end{aligned}$$

Figure 5: Comparison of the nonstandard semantics for a conditional expression with label p and internal labels ip_0 and ip_1

an additional argument. It is ignored when performing 0CFA, and it is stored as the contour counter in 1CFA.

This generalizes to *k*CFA, where the last k call sites are tracked by the abstract contour counter. The more information we store in the abstract contour counters, the more detailed our analysis will be, but also more expensive to calculate.

Because semantic values $\widehat{\mathrm{D}}$ are now sets of procedures, when evaluating a call expression $(f\ a_1 \ldots a_n)$, there is a set of possible procedures that f can evaluate to, and $\widehat{\mathcal{F}}$ will be called for each of them. The resulting call caches need to be joined by pointwise set union. Similarly, the continuations of primitive operations are sets of procedures, for all of which $\widehat{\mathcal{F}}$ is called and the results are joined, as seen in Figure 5.

2.5 Main results

These main results about the semantics functions are stated and shown in Shivers' dissertation:

1. The call cache returned by the exact semantics is indeed a partial map,

2. the abstract semantics approximates the exact semantics and
3. the abstract semantics is computable.

But before starting to prove these results, Shivers establishes that the recursive definitions he gave for $\mathcal{F}$ and $\mathcal{C}$ (and by analogy, $\widehat{\mathcal{F}}$ and $\widehat{\mathcal{C}}$) are sound. He proceeds in the standard way of turning a recursive definitions $f = Ff$ for a function $f \in X$, where X is a function space with a chain-complete partial order, into a functional $F\colon X \to X$. F is then shown to be continuous and therefore has a least fixed point, which serves as the definition of f.

The need to show that the result of $\mathcal{PR}$ is a partial map arises from the fact that, for an easier proof of continuity, Shivers changes the answer domain to be

$$\text{Ans} = \mathcal{P}((\text{LAB} \times \text{BEnv}) \times \text{Proc}).$$

The actual proof is only outlined in his dissertation.

In contrast, the proof that $\widehat{\mathcal{PR}}$ safely approximates $\mathcal{PR}$ is given in full detail. For any of the types Proc, D, D^*, BEnv, VEnv, CN and CCache, he defines an abstraction function (written $|\cdot|$) from the exact type to the corresponding abstract type. He also defines partial orders on the abstract types (here written $\cdot \lessapprox \cdot$), expressing that one value is more specific than another value. The main result is then

> **Theorem 6**: For any program l, call site label c and binding environment β, we have
>
> $$|(\mathcal{PR}\ l)\ (c, \beta)| \lessapprox (\widehat{\mathcal{PR}}\ l)(c, |\beta|).$$

The largest part of the proof is to show two similar statements relating $\mathcal{F}$ and $\widehat{\mathcal{F}}$ resp. $\mathcal{C}$ and $\widehat{\mathcal{C}}$. Because these are mutually recursive, both statements are shown in one big step.

The machinery to prove results about functions defined as fixed points is the fixed point induction: A predicate P holds for the least fixed point of F if the following three conditions are true:

- P is an admissible predicate. This is closely related to continuity and states that if P holds for each element of an infinite chain, it holds for the limit of the chain.
- $P\ \bot$ holds.
- If $P\ f$ is true, then $P\ (F\ f)$ is true.

The computability result is not shown explicitly for $\widehat{\mathcal{PR}}$, but rather for general recursive equations of the shape

$$f\ x = g\ x \cup f\ (r\ x)$$

where each function invocation makes a local contribution $g\ x$ to the result and then recurses with a new argument $r\ x$. Shivers shows that in that case, the least fixed point is given by

$$f\ x = \bigcup_{i=0}^{\infty} g\ (r^i\ x).$$

If additionally the argument space is finite, the infinite union is actually finite as well and the calculation is complete after a finite number of iterations.

The semantics functions $\widehat{\mathcal{F}}$ and $\widehat{\mathcal{C}}$ are nearly of this type, but they have a branch factor greater than one. Thus, for solution to an equations like

$$f\ x = g\ x \cup \bigcup \{f\ x' \mid x' \in R\ x\}$$

he introduces the *powerset relative* of a function h as $\underline{h}\ X \coloneqq \{h\ x \mid x \in X\}$ and transforms the above equation to

$$\underline{f}\ X = \underline{g}\ x \cup \underline{f}\ (\underline{R}\ Y)$$

to be able to apply the previous result.

The required steps to apply this result to the mutually recursive definitions $\widehat{\mathcal{F}}$ and $\widehat{\mathcal{C}}$ are not explained in the dissertation.

2.6 Example

In Figure 6 on the following page lists a functional program in continuation-passing style, calculating the sum of the first ten natural numbers. Its code positions are subscripted by labels which are automatically assigned by the Haskell code presented in the following chapter. Lambda expressions have labels 1, 3, 8, 13, 18, and 24. Call expressions, where the label is placed below the space before the callee, are 2, 4, 9, 14, 19, 25 and 28. Labels 5 and 6 denote the two internal call sites of the conditional expression, while labels 10 and 15 denote the internal call sites of the two primops `(+)`.

```
(λ cont.
 1
   let rec = (λ p i c'.
 2            3
                if i
              4
                then (λ . (+) p i (λ p'. (+) i -1 (λ i'. rec p' i' c')))
                5     8  9 10      13   14 15       18   19
                else (λ . c' p))
                6     24 25
   in rec 0 10 cont)
     28
```

Figure 6: CPS program calculating the sum of the first ten natural numbers

A contour environment specifies for each binding position which binding, identified by a contour counter (in this section printed in italics to distinguish them from syntactical labels), is currently in place. The `let`-expression in the above code will store the lambda expression with label 3 in the variable environment, coupled with the contour environment

$$\{1 \mapsto \mathit{0},\ 2 \mapsto \mathit{1}\}$$

indicating that the variable `cont` should resolve to the value bound when the contour counter was *0* and that `rec` should resolve to the value bound when the contour counter was *1*. The closure is called at label 28, which adds the following entry to the call cache calculated by the exact semantics

$$(28,\ \{1 \mapsto \mathit{0},\ 2 \mapsto \mathit{1}\}) \mapsto (3,\ \{1 \mapsto \mathit{0},\ 2 \mapsto \mathit{1}\}).$$

The lambda expression will store the values passed as arguments using the contour counter *2* and passes the following, extended binding environment to call 4:

$$\{1 \mapsto \mathit{0},\ 2 \mapsto \mathit{1}, 3 \mapsto \mathit{2}\}.$$

The next call of lambda 3, from position 19, is recorded in the exact call cache by

$$(19,\ \{1 \mapsto \mathit{0},\ 2 \mapsto \mathit{2},\ 3 \mapsto \mathit{2},\ 8 \mapsto \mathit{4},\ 13 \mapsto \mathit{6},\ 18 \mapsto \mathit{8}\}) \mapsto (3,\ \{1 \mapsto \mathit{0},\ 2 \mapsto \mathit{1}\}).$$

Now, the arguments will be stored using the contour counter *9* and call 4 is evaluated with a binding environment of

$$\{1 \mapsto \mathit{0},\ 2 \mapsto \mathit{1}, 3 \mapsto \mathit{9}\},$$

$4 \mapsto \{5\}$	$14 \mapsto \{15\}$
$5 \mapsto \{8\}$	$15 \mapsto \{18\}$
$6 \mapsto \{24\}$	$19 \mapsto \{3\}$
$9 \mapsto \{10\}$	$25 \mapsto \{\mathit{Stop}\}$
$10 \mapsto \{13\}$	$28 \mapsto \{3\}$

Figure 7: 0CFA-analysis of the example program

ensuring that the correct values of `p`, `i` and `c'` will be obtained when looking up these variables in the variable environment.

The complete exact call cache for this program contains 74 entries and we abstain from reproducing it here. The abstract call cache obtained using the 0CFA abstraction is given in Figure 7. Since the contour environments carry no information in this case, they are omitted. Note that the label 5 on the right hand side of the edge $4 \mapsto \{5\}$ represents the whole if expression, not just the first branch.

Applying the 1CFA abstraction, we obtain the call cache in Figure 8 on the following page. The contour counter -1 represents the context "outside" the analyzed program. The left hand sides of the entries, as well as closures on the right hand side, now carry their abstracted contour environment. In this example, it allows us to tell evaluations of the lambda expression 3 called from position 28 apart from those called from position 19. In our case, the gain is small as in either case we call the same procedures.

But consider the following lambda expression, which implements the function composition operator $\circ$ in continuation-passing style by evaluating `g`, feeding the result to `f` and passing evaluation on to the continuation `c`:

```
λ f g c .  g (λ r.  f r c)
```

If this function is used in two totally different contexts, a 0CFA analysis could not tell the two apart and would add return edges from $\circ$ to both of them. 1CFA would annotate those edges by the context that $\circ$ had been called in, and thus prevents mixing these different control flows.

But 1CFA cannot help if a commonly used function `f` itself calls another function `g` which then calls the continuation, as the context of `g` is always within `f` and `f`'s context is lost. In that example, 2CFA would be required, and there is no general solution to this problem.

$$(4, \{1 \mapsto -1,\ 2 \mapsto 2,\ 3 \mapsto 19\}) \mapsto \{5\}$$
$$(4, \{1 \mapsto -1,\ 2 \mapsto 2,\ 3 \mapsto 28\}) \mapsto \{5\}$$
$$(5, \{5 \mapsto 4\}) \mapsto \{(8, \{1 \mapsto -1,\ 2 \mapsto 2,\ 3 \mapsto 28\})\}$$
$$(6, \{6 \mapsto 4\}) \mapsto \{(24, \{1 \mapsto -1,\ 2 \mapsto 2,\ 3 \mapsto 28\})\}$$
$$(9, \{1 \mapsto -1,\ 2 \mapsto 2,\ 3 \mapsto 19,\ 8 \mapsto 5\}) \mapsto \{10\}$$
$$(9, \{1 \mapsto -1,\ 2 \mapsto 2,\ 3 \mapsto 28,\ 8 \mapsto 5\}) \mapsto \{10\}$$
$$(10, \{10 \mapsto 9\}) \mapsto \{(13, \{1 \mapsto -1,\ 2 \mapsto 2,\ 3 \mapsto 28,\ 8 \mapsto 5\})\}$$
$$(14, \{1 \mapsto -1,\ 2 \mapsto 2,\ 3 \mapsto 19,\ 8 \mapsto 5,\ 13 \mapsto 10\}) \mapsto \{15\}$$
$$(14, \{1 \mapsto -1,\ 2 \mapsto 2,\ 3 \mapsto 28,\ 8 \mapsto 5,\ 13 \mapsto 10\}) \mapsto \{15\}$$
$$(15, \{15 \mapsto 14\}) \mapsto \{(18, \{1 \mapsto -1,\ 2 \mapsto 2,\ 3 \mapsto 28,\ 8 \mapsto 5,\ 13 \mapsto 10\})\}$$
$$(19, \{1 \mapsto -1,\ 2 \mapsto 2,\ 3 \mapsto 19,\ 8 \mapsto 5,\ 13 \mapsto 10,\ 18 \mapsto 15\}) \mapsto \{(3, \{(8, \{1 \mapsto -1,\ 2 \mapsto 2\})\}$$
$$(19, \{1 \mapsto -1,\ 2 \mapsto 2,\ 3 \mapsto 28,\ 8 \mapsto 5,\ 13 \mapsto 10,\ 18 \mapsto 15\}) \mapsto \{(3, \{(8, \{1 \mapsto -1,\ 2 \mapsto 2\})\}$$
$$(25, \{1 \mapsto -1,\ 2 \mapsto 2,\ 3 \mapsto 19,\ 24 \mapsto 6\}) \mapsto \{\mathit{Stop}\}$$
$$(25, \{1 \mapsto -1,\ 2 \mapsto 2,\ 3 \mapsto 28,\ 24 \mapsto 6\}) \mapsto \{\mathit{Stop}\}$$
$$(28, \{1 \mapsto -1,\ 2 \mapsto 2\}) \mapsto \{(3, \{1 \mapsto -1,\ 2 \mapsto 2)\})\}$$

Figure 8: 1CFA-analysis of the example program

CHAPTER 3

The Haskell Prototype

To get a better understanding of the algorithms, we first implemented a prototype in Haskell. As the semantics are given in denotational style, they are easily translated into a functional program. The elements of the syntax definition of the CPS style language are directly turned into data type definitions, given in Figure 9 on the next page.

At this point, we slightly deviate from the original: Shivers distinguishes the sets FUN and ARG as direct sums (cf. Figure 1 on page 10) and later defines the function $\mathcal{A}$ on the union of FUN and ARG (cf. Figure 2 on page 11). This would be tedious and unwieldy to express in the Haskell type system, therefore we use one set VAL instead of FUN and ARG, which does not affect the algorithms in any notable way:

$$\text{VAL} ::= \text{LAM} + \text{REF} + \text{PRIM} + \text{CONST}$$

We also drop the special constant `#f` representing false from the syntax and the corresponding value false from the set of semantic values, which are now just the integers. The `if` primitive operation considers zero as false and any other integer as true.

3.1 Type system tricks

As mentioned in the last chapter, most elements of the syntax carry a label. These labels need to be unique and every mention of a variable needs to carry the label of the variable's binding position. To prevent the hypothetical user

```
type Prog = Lambda
newtype Label = Label Integer
data Var = Var Label String
data Lambda = Lambda Label [Var] Call
data Call = App Label Val [Val]
          | Let Label [(Var, Lambda)] Call
data Val = L Lambda
         | R Label Var
         | C Label Const
         | P Prim
type Const = Integer
data Prim = Plus Label
          | If Label Label
```

Figure 9: The Haskell representation of abstract syntax trees

of the library from getting the labels wrong, we exploit the *smart constructor* idiom.

A type alias marks not-fully-constructed syntax tree values and prevents their use in other functions.

```
newtype Inv a = Inv { unsafeFinish :: a } deriving (Show, Eq)
```

The deconstructor `unsafeFinish :: Inv a → a` is only to be used internally.

Now a value of type `Inv Lambda` represents a lambda expression without correct labels. For each constructor there is a function (the smart constructor), taking the same arguments except the label, and building the unfinished value. We named it after the constructor it replaces, written lower case. Because **let** and **if** are keywords in Haskell, the corresponding smart constructors are called `let_` and `if_`. For example, the smart constructor for lambda expressions is defined as

```
lambda :: [Inv Var] → Inv Call → Inv Lambda
lambda vs (Inv c) = Inv $ Lambda noProg (map unsafeFinish vs) c
```

where `noProg` throws, if it were evaluated, an exception:

```
noProg :: a
noProg = error "Smart constructors used without calling prog"
```

```
-- Returns the sum of the first 10 natural numbers
ex3 :: Prog
ex3 = prog $ lambda ["cont"] $
        let_ [("rec", lambda ["p", "i", "c'"] $
                        app if_
                            [ "i"
                            , l $ lambda [] $
                                app plus ["p", "i",
                                    l $ lambda ["p'"] $
                                        app plus ["i", -1,
                                            l $ lambda ["i'"] $
                                                app "rec" [ "p'", "i'", "c'" ]
                                            ]
                                    ]
                            , l $ lambda [] $
                                app "c'" ["p"]
                            ]
        )] $ app "rec" [0, 10, "cont"]
```

Figure 10: Example code, defined using smart constructors

There is only one function for public consumption that removes the Inv-wrapper, with signature prog :: Inv Lambda → Prog. It uses a state monad to hand out unique labels and keeps track of variable bindings to correctly set the reference to the binding position.

For additional convenience, we made the types Var, Val and Inv a instances of the IsString type class which allows the compiler, if the language extension OverloadedStrings is enabled, to turn string literals into values of these kinds. Using the same trick, a **Num**-instance for Val allows to enter constants directly as literal numbers.

A simple example program, returning the result of the worldshaking calculation $1 + 1$, can now entered as

```
ex2 :: Prog
ex2 = prog $ lambda ["cont"] $
        app plus [1, 1, "cont"]
```

instead of the much verbose

```
(λ cont.
   let rec = (λ p i c'.
                if i
                then (λ . (+) p i (λ p'. (+) i -1 (λ i'. rec p' i' c')))
                else (λ . c' p))
   in rec 0 10 cont)
```

Figure 11: Pretty-printed output

```
ex2 :: Prog
ex2 = Lambda (Label 1) [Var (Label 1) "cont"] $
        App (Label 2) (P (Plus (Label 3))) [
                C (Label 4) 1,
                C (Label 5) 1,
                R (Label 6) (Var (Label 1) "cont")].
```

A larger example is included in Figure 10 on the preceding page, where the first ten natural numbers are added up.

3.2 Pretty Printing

Another feature of the Haskell prototype is pretty-printing of an abstract syntax tree. It uses the standard pretty printing library of Haskell [5] and renders the above example to the string "`(λ cont. (+) 1 1 cont)`". Figure 11 shows the pretty printed presentation of the program in Figure 10 on the preceding page.

Also, a conversion function is included that renders a program in the syntax used by Isabelle, to make it more convenient to obtain example programs within Isabelle.

CHAPTER 4

The Isabelle Formalization

THE main mission of this student research project was to implement Shivers' algorithms in the theorem prover Isabelle[2] and to prove his main results as introduced in Chapter 2. Shivers was already very rigorous and formal in his proofs, which helped a lot in the course of this project. The main hurdles to take were

- representing partial functions, such as the semantics functions, in Isabelle within the default logic HOL, which is a logic of total functions, and
- obtaining the fact that the set of subterms of a program is finite.

4.1 Structure

I separated the formalizations into several files (called theories in Isabelle-speak). Their dependencies are given in Figure 12 on the next page, where theories in rectangles contain definitions, boxed theories contain the main results and theories in rounded rectangles contain auxiliary definitions and lemmas. The definitional theories have directly corresponding counter-parts in the Haskell prototype. The following list gives a quick summary of their content.

CPSScheme contains the definitions of the abstract syntax to represent the functional programs. The types (Figure 13 on page 27) are very similar to those in the Haskell prototype (Figure 9 on page 22).

Eval is an implementation of the standard semantics.

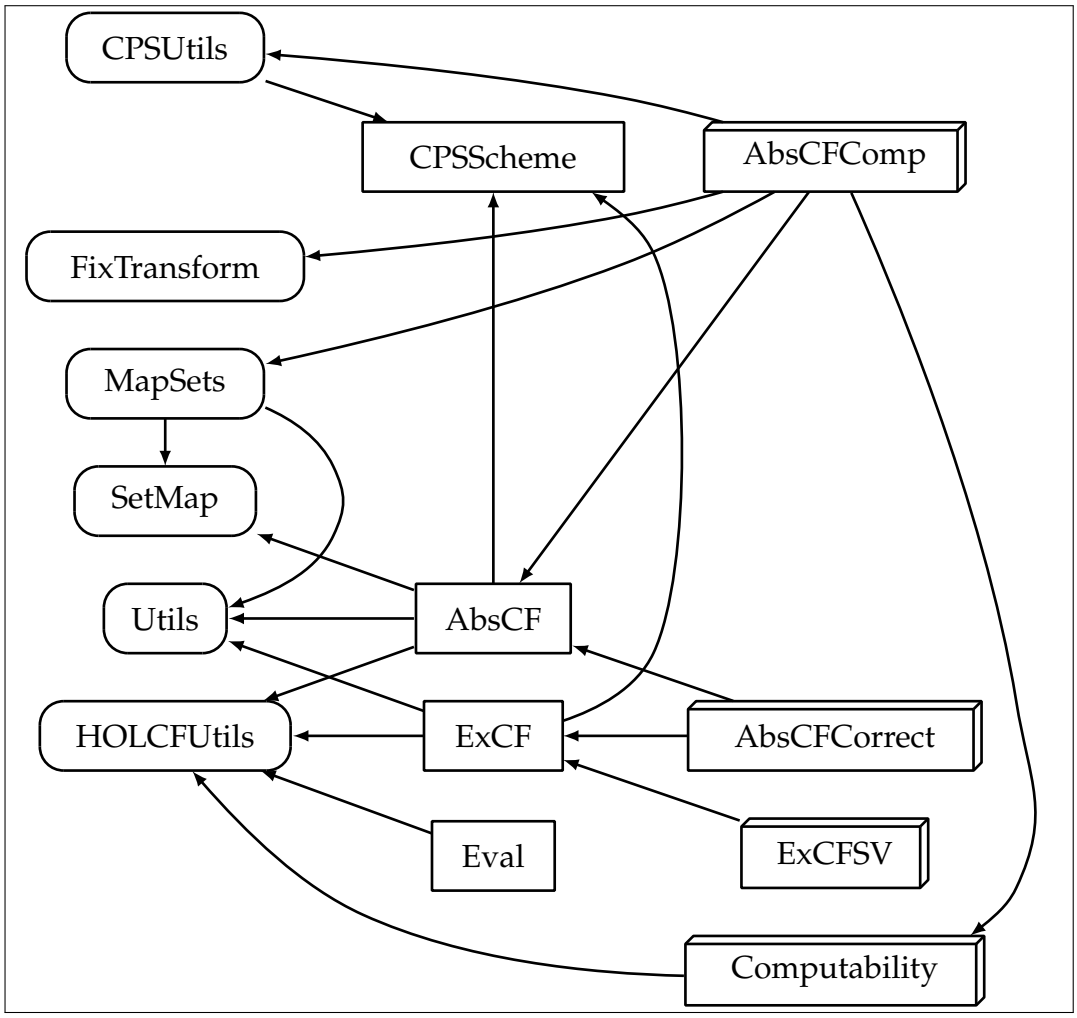

Figure 12: Isabelle theories and their dependencies

```
types label = nat

types var = label × string

datatype prim = Plus label | If label label

datatype lambda = Lambda label var list call
  and call = App label val val list
       | Let label (var × lambda) list call
  and val = L lambda | R label var | C label int | P prim

types prog = lambda
```

Figure 13: Isabelle data types

ExCF is an implementation of the exact nonstandard semantics.
AbsCF is an implementation of the abstract nonstandard semantics.

ExCFSV proves that a call cache returned by the exact nonstandard semantics is indeed a partial map.
AbsCFCorrect proves that the abstract nonstandard semantics safely approximate the exact nonstandard semantics.
Computability contains Shivers' general treatment of equations whose least solution is computable.
AbsCFComp applies the results from the previous theory to the abstract nonstandard semantics. This step was skipped by Shivers.

Utils is a potpourri of various lemmas not specific to our project, some of which could very well be included in the default Isabelle library.
HOLCFUtils contains generic lemmas related to the use of *HOLCF*, a domain theory extension to Isabelle.
CPSUtils defines sets of subterms for programs and proves their finiteness.
FixTransform transforms fixed point expressions defining two mutually recursive functions to fixed point expressions defining a single function.
SetMap contains functions and lemmas to work with set-valued maps.
MapSets defines sets of functions and sets of maps, and shows the finiteness of such maps, if their domains and ranges are finite.

4.2 Domain theory in Isabelle

Generally it is straight-forward to transform functional code, such as the Haskell prototype, into Isabelle. For total functions, the **function** package[7] provides a great deal of convenience, such as automatic termination proofs, overlapping patterns in the defining equations and mutual recursion. Unfortunately, the important functions that we would like to define, $\mathcal{F}$ and $\mathcal{C}$, are not total functions: For non-terminating programs they recurse endlessly.

The function package has some support for partial functions using the *domintros* option, which introduces a termination predicate that then appears in the premises to any lemma about the functions. This turned out to be a major restriction when we formalized the functions in that setting and we looked for alternatives.

Shivers handles the issue using the machinery of domain theory, where functions defined by recursive definitions are obtained by constructing a functional on the space of function, proving that it is continuous and then taking the least fixed point of the function as the desired definition. The *HOLCF*-package[8], which is an extension to the standard Higher Order Logic (HOL) of Isabelle, provides the necessary definitions to work with domain theory in Isabelle.

A simple example for a function defined by recursion would be the function $f\colon \mathbb{N} \to \mathbb{N}$ that gives final value in the Collatz series starting with its argument:

$$f(n) \coloneqq \begin{cases} 1, & \text{if } n = 1 \\ f(\frac{n}{2}), & \text{if } n \text{ is even} \\ f(3 \cdot n + 1), & \text{otherwise.} \end{cases}$$

Defining this function with the **function** package would be very difficult, as we either had to prove that it is total, thereby proving the Collatz conjecture, or work with the inconvenient domain predicates generated by the *domintros* option.

Within *HOLCF*, we can define the function f as the least defined function fulfilling the above property. To be able to do so, the function space needs to have a chain-complete partial order. We therefore extend the range of the function by the special value $\bot$ to indicate undefinedness. Now f is the

```
theory Collatz imports HOLCF begin

fixrec f :: nat discr → nat lift
where f·n = (if undiscr n = 1 then
          Def 1
        else if even (undiscr n) then
          f·(Discr ((undiscr n) div 2))
        else
          f·(Discr (3 * undiscr n + 1)))

lemma f·(Discr 42) = Def 1 by simp
end
```

Figure 14: Function definitions with *HOLCF*

least fixed point $f = \mathrm{fix}(F)$ under a functional $F\colon (\mathbb{N} \to \mathbb{N}_\perp) \to (\mathbb{N} \to \mathbb{N}_\perp)$ derived from the above specification:

$$F(f) \coloneqq \lambda n.\ \begin{cases} 1, & \text{if } n = 1 \\ f(\frac{n}{2}), & \text{if } n \text{ is even} \\ f(3 \cdot n + 1), & \text{otherwise.} \end{cases}$$

The least fixed point exists and is unique if F is continuous, i.e. if it is monotonous and preserves limits of chains.

HOLCF relieves the user from the burden of transforming the recursive equations into a functional by offering the **fixrec** command, which is demonstrated in Figure 14. This command turns a recursive function definition or several mutually recursive function definitions internally into a functional, defines the function as the fixed point and proves simplification and induction lemmas for the function.

By default, no partial order is defined for $\mathbb{N}$, because there is more than one sensible choice. To tell the system which order to use we use the wrapper *discr* for the discrete ordering, with conversion functions *Discr* and *undiscr*, and *Lift* for the ordering with one additional element denoting bottom, with constructor *Def*.

A lemma such as $f(42) = 1$ would not be shown as easy as in Figure 14 if we had used the **function** package with the *domintros* option, as the termination predicate had to be proven first.

Clearly, *HOLCF* is the right choice to define our semantics function. In this case, the domain is a set. In Isabelle, sets over a type *'a* are just functions *'a* $\Rightarrow$ *bool*. In the theory *HOLCFUtils*, we therefore defined a partial order on *bool*, with *false* $\sqsubset$ *true*, to obtain the usual order on sets, where $A \sqsubseteq B \iff A \subseteq B$. For the integers, which occur as labels in the return value, contour counters and types from the syntax definition, the discrete order is defined. The arguments to the $\mathcal{C}$ and $\mathcal{F}$ functions are uncurried, i.e. written as one quadruple, and wrapped in *Discr*. This way, all occurring types have a partial order and *HOLCF* can work with them.

As mentioned before, least fixed points exist if the functional is continuous. The **fixrec** command hides this from the user by automatically proving continuity if the functional is defined using continuous building blocks. If this is not sufficient, an error message is shown which contains the remaining continuity goal and the user can add new rules to the set used by **fixrec**, named *cont2cont*. In our case, continuity lemmas for booleans and sets were helpfully provided by Brian Huffman, and we wrote continuity lemmas about case expressions for our custom data types.

4.3 Fixed point induction

The main method to show results about a least fixed point is the fixed point induction, as explained on page 16. This technique was used both to obtain that the exact semantics return a partial map and that the abstract semantics safely approximate the exact semantics. Again, *HOLCF* provides some machinery to show the admissibility criterion without much effort.

The Isabelle formalization of the correctness result, Lemma 6, and the two main stepping stones to prove it, Lemmas 8 and 9, are printed in Figure 15. These two lemmas need to be proven at once due to the mutual recursive definitions of $\mathcal{F}$ and $\mathcal{C}$ resp. $\widehat{\mathcal{F}}$ and $\widehat{\mathcal{C}}$. The lemma relates two fixed points: One for the exact semantics and one for the abstract semantics. Shivers proceeds by parallel fixed point induction, which, in order to relate fixed points of two functionals F and G, requires relating $\bot$ and $\bot$, then assuming that f and g relate and showing that $F\ f$ and $G\ g$ relates. In our case of the approximation relation, this can be simplified: We use fixed point induction for fixed point of the exact semantics function and hold the abstract semantics fixed. The proof proceeds by explicit case analysis of the argument vectors of $\mathcal{F}$ and $\mathcal{C}$, *fstate* and *cstate*.

lemma *lemma89*:
fixes *fstate-a* :: *'c*::*contour-a* $\widehat{fstate}$ **and** *cstate-a* :: *'c*::*contour-a* $\widehat{cstate}$
shows $|fstate| \lessapprox fstate\text{-}a \Longrightarrow |\mathcal{F}\cdot(Discr\ fstate)| \lessapprox \widehat{\mathcal{F}}\cdot(Discr\ fstate\text{-}a)$
and $|cstate| \lessapprox cstate\text{-}a \Longrightarrow |\mathcal{C}\cdot(Discr\ cstate)| \lessapprox \widehat{\mathcal{C}}\cdot(Discr\ cstate\text{-}a)$

lemma *lemma6*: $|\mathcal{PR}\ l| \lessapprox \widehat{\mathcal{PR}}\ l$

Figure 15: The final correctness result and the main lemma to proof it

A disadvantage of the fixed point induction is that it is not possible to use auxiliary lemmas about the function: While proving the inductive case, one does not show results for the function in question, but for an information-theoretical approximation. Thus, any previously shown results are not available. Therefore, the inductions of the auxiliary lemmas have to be intertwined with the induction of the main result.

In our case, we had to resort to this measure in the proof that the call cache returned by the exact semantics is a partial map, in theory *ExCFSV*: The auxiliary lemma stated that if b is the contour pointer passed to $\mathcal{F}$ resp. $\mathcal{C}$, then every contour environment in the returned call cached mentions a contour counter greater or equal to b, and inductively assumes that for $\mathcal{F}$, b is strictly larger than all contour counters occurring in the arguments to $\mathcal{F}$ and for $\mathcal{C}$ the contour counter b occurs in the contour environment passed to C and is larger than all contour counters occurring in the arguments to $\mathcal{C}$. The main lemma just states that the call cache is a partial map, which corresponds to the predicate *single-valued* from the Isabelle library. The resulting intertwined lemma, as defined in Isabelle, is shown in Figure 16 on the next page.

The proof itself again proceeds by case-analysis of the arguments. Each case is handled explicitly, as the automation present in Isabelle could not solve the goals directly.

4.4 Finiteness of subexpressions

For the computability proof, we need the fact that the set of subexpressions of a program is finite. This lemma, although obvious, turned out to be tricky to prove. One complication arises from the fact that there are subexpressions of various types in a program – calls, lambdas, variables, values, labels, primitive operations – some of which are mutually recursive.

```
lemma cc-single-valued':
    ⟦ ∀ b' ∈ contours-in-ve ve. b' < b
    ; ∀ b' ∈ contours-in-d d. b' < b
    ; ∀ d' ∈ set ds. ∀ b' ∈ contours-in-d d'. b' < b
    ⟧
    ⟹
    (   single-valued (F·(Discr (d,ds,ve,b)))
    ∧ (∀ ((lab,β),t) ∈ F·(Discr (d,ds,ve, b)).
          ∃ b'. b' ∈ ran β ∧ b ≤ b')
    )
 and ⟦ b ∈ ran β'
    ; ∀ b'∈ran β'. b' ≤ b
    ; ∀ b' ∈ contours-in-ve ve. b' ≤ b
    ⟧
    ⟹
    (   single-valued (C·(Discr (c,β',ve,b)))
    ∧ (∀ ((lab,β),t) ∈ C·(Discr (c,β',ve,b)).
          ∃ b'. b' ∈ ran β ∧ b ≤ b')
    )

lemma evalPR-single-valued:
  single-valued (PR prog)
```

Figure 16: Intertwined fixed point inductions and the final main result

Our first approach was to define the set of subexpressions by a recursive function that calls itself for the immediate subexpressions of the argument, joins the results and inserts the argument. Using the induction rule for this function, the finiteness of the resulting set is easily shown. Unfortunately, we also need lemmas about how these sets relate: If we have a lambda expression and know that it is in the set of lambdas of a program, then we need to know that the call within the lambda expression is in the set of calls of this program. Because the two sets are generated independently, this lemma required a full-fledged induction over the syntax tree. The induction rule for the mutually recursive types *lambda*, *call* and *val* also requires special hypotheses for the occurrences of the types which are wrapped in lists, such as the list of bindings in a *Let* expression. So although we are only interested in the seemingly trivial fact *Lambda l vs c* $\in$ *lambdas x* $\Longrightarrow$ *c* $\in$ *calls x*, the complicated proof shown in Figure 17 on page 34 is required. We had to add 12 such lemmas, which fortunately were all provable by only slight variations of the apply script in the figure.

A much cleaner approach would be a combined definition of the sets of subexpressions in one **inductive-set**, which could be defined by exactly the 12 lemmas mentioned above. The downside of that approach is that the finiteness of these sets turned out to be hard to come by.

Under the hood **inductive-set** works very similar to **fixrec** as it defines the resulting set as the least fixed point of a functional. Because sets form a complete lattice, only monotonicity of the functional is required and proven automatically. Our second approach was now to create a generally applicable lemma where such a functional F gives rise to a finite set. The conditions for finiteness are

- Monotonicity of the functional, which can be proven mechanically.
- Finiteness preservation, i.e. if S is finite, then $F\ S$ is also finite.
- Descending measure of new elements: Each element $x \in FS \setminus S$ is either added unconditionally ($x \in F\ \{\}$), or there is an element $y \in S$ such that $s(x) < s(y)$ for some natural-valued measure function s (usually Isabelle's *size* function).

We proved this lemma but then ran into a dead end when we found out that the implementation of the **inductive-set** command treats mutually recursive sets by constructing a single intermediate fixed point and defining each set as a projection thereof. The construction happens to be such that the large set is infinite, preventing our approach from succeeding. The infinite fixed point could have been avoided by modifying the construction within **inductive-set** slightly.

A third approach was suggested to us by Andreas Lochbihler. Here, subexpressions of any type are assigned a position, which is a list of natural numbers. The set *Pos p* of valid positions in the program p is defined using **function** and finiteness is shown easily and automatically. A partial function *subterm* of type

$$lambda + call + val \Rightarrow pos \rightharpoonup lambda + call + val$$

is defined, mapping a valid position to the corresponding subexpression. We show that the inductively defined sets of subexpressions are subsets of the range of *subterm*, by an explicit inductive proof, and thus obtained the finiteness of these sets. This approach required about 40% more lines of code for the sets of lambdas, calls and values (which are the mutually recursive ones) than the first approach, but is cleaner and would scale better.

```
lemma
  fixes list2 :: (var × lambda) list and t :: var×lambda
  shows lambdas1: Lambda l vs c ∈ lambdas x ⟹ c ∈ calls x
  and Lambda l vs c ∈ lambdasC y ⟹ c ∈ callsC y
  and Lambda l vs c ∈ lambdasV z ⟹ c ∈ callsV z
  and ∀z∈ set list. Lambda l vs c ∈ lambdasV z ⟶ c ∈ callsV z
  and ∀x∈ set (list2. Lambda l vs c ∈ lambdas (snd x) ⟶ c ∈ calls (snd x)
  and Lambda l vs c ∈ lambdas (snd t) ⟹ c ∈ calls (snd t)
apply (induct rule:lambda-call-val.inducts)
apply auto
apply (case-tac c, auto)[1]
apply (rule-tac x=((a, b), ba) in bexI, auto)
done
```

Figure 17: One of 12 unwieldy proofs to relate two sets of subexpressions

4.5 Finishing the computability proof

As mentioned on page 17, Shivers shows the computability only abstractly for a single recursively defined function, and leaves it to the reader to generalize this to mutually recursive functions. We carried this step out in detail. To do so, we need to transform a fixed point for two functions (implemented in *HOLCF* as a fixed point over a tuple) to a simple fixed point equation. The approach here works as long as both functions in the tuple have the same return type, using the isomorphisms

$$\begin{aligned}(A \to X) \times (B \to X) &\rightleftarrows (A + B) \to X \\ (f, g) &\mapsto \left(x \mapsto \begin{cases} f\ a, & \text{if } \iota_A\ a = x \\ g\ b, & \text{if } \iota_B\ b = x \end{cases} \right) \\ (h \circ \iota_A, h \circ \iota_B) &\leftarrow\!\shortmid h\end{aligned}$$

where $\iota_A \colon A \to A + B$ and $\iota_B \colon B \to A + B$ are the injection functions.[1]

In theory *FixTransform* we showed that a fixed point can be transformed using any retractable continuous function:

$$g \circ f = \mathrm{id} \implies \mathrm{fix}(F) = g(\mathrm{fix}(f \circ F \circ g))$$

[1] Interestingly, by using the familiar notation X^A for the set $A \to X$, this corresponds to the well-known law about exponentiation $X^A \cdot X^B = X^{A+B}$. In fact, this holds in any bicartesian closed category. But now I am talking abstract nonsense.

lemma *args-finite*:
$\mathit{finite}\ (\bigcup i.\ \mathit{iterate}\ i \cdot (\underline{\mathit{abs\text{-}R}}) \cdot \{\mathit{initial\text{-}r}\ \mathit{prog}\})$

lemma *a-evalCPS-iterative*:
$\widehat{\mathcal{PR}}\ \mathit{prog} = \underline{\mathit{abs\text{-}g}} \cdot (\bigcup i.\ \mathit{iterate}\ i \cdot (\underline{\mathit{abs\text{-}R}}) \cdot \{\mathit{initial\text{-}r}\ \mathit{prog}\})$

Figure 18: Computability results as expressed in Isabelle

and used this with the functions that convert between the tuple of functions and the combined function to transform fixed points as required. In theory *AbsCFComp*, this these results are applied to $\widehat{\mathcal{C}}$ and $\widehat{\mathcal{F}}$ and combined with Shivers abstract computability lemmas. We obtain the two lemmas given in Figure 18, where *initial-r* is the initial argument to $\widehat{\mathcal{F}}$ as set up in $\widehat{\mathcal{PR}}$, *abs-R* gives, for an argument, the set of arguments the semantics functions recurse to and *abs-g* calculates the call cache entries for one such argument.

4.6 Cosmetics

We tried to follow Shivers very closely not only in substance but also in presentation. Isabelle is already good in generating documents from its theories that resemble common mathematics notation very closely. We employed some tricks to increase similarity with Shivers' dissertation.

Functions in Isabelle need to have an alphanumeric name. Shivers calls his function by symbols ($\mathcal{F}$, $\widehat{\mathcal{C}}$,...). To achieve this, we assign the symbol a syntax translations. So the evaluation function for lambdas is called *evalF*, but Isabelle also understands the code `\<F>` which is by default set up to render as $\mathcal{F}$. We use the alternative symbol throughout the code and real name *evalF* only appears in the definition and when referencing lemmas generated by **function**.

Shivers uses some symbols that do not exist as predefined Isabelle symbols, e.g. $\widehat{\mathcal{F}}$ or $\widehat{\mathcal{PR}}$. In that case, we "invent" the symbol by writing `\<aPR>` and adding an appropriate definition for the LaTeX command `\isasymaPR` to our document.

A similar definition is used for the power-set function used in the computability proof. Shivers denotes it by underlining the argument. This was achieved by a syntax translation `\<^ps>` and a corresponding LaTeX command `\isactrlps`, defined to be {`\uline #1`}.

In the proof about the safe approximation of $\widehat{\mathcal{F}}$ and $\widehat{\mathcal{C}}$, two symbols are used overloaded: The abstraction functions $|\cdot|$ and the approximation relations $\lessapprox$. This is per se not a problem, as Isabelle allows for ambiguous syntax translations. In that case, it generates more than one parse tree and picks the (hopefully unique) tree that typechecks.

Unfortunately, this does not work well in our case: There are eight concrete functions which we want to write as $|\cdot|$ and some expressions have multiple occurrences of these, causing an exponential blow-up of combinations.

Luckily, the latest development version of Isabelle contains a module by Christian Sternagel and Alexander Krauss for ad-hoc overloading, where the choice of the concrete function is done at parse time and immediately, based on the argument types. Using this system, we were able to write $|\cdot|$ and $\lessapprox$ just as Shivers did.

4.7 Development Environment

The most common development environment for working with Isabelle theories is ProofGeneral, an extension to the text editor Emacs. Unless one is already familiar with Emacs, learning it in addition to Isabelle might not be desired.

One of the alternatives is the relatively young project i3p[4], which provides a similar feature set based on the Netbeans frameworks and thus offers a fairly standard user experience. Over the course of the project, we have found and reported some minor bugs and reported these to the author, who then released new versions.

A more severe problem was observed with proofs involving large goal state: i3p, in contrast to ProofGeneral, reads the goal state after each command to allow the user to read earlier goal states without having to re-evaluate the theory. This is a noticeable improvement of usability. Unfortunately, it is quite expensive to print large Isabelle goal states, as they are run through various stages of syntax translations. Some theories become unbearable slow to evaluate and in some cases, we employed workarounds to avoid large goal states. But also for this issue the author found a solution and the latest version of i3p now uses lazy goal states, which means that goal states are only generated by Isabelle when they are actually displayed by i3p. The overhead for remembering the earlier goal states within Isabelle is low, thanks to the efficient sharing of values in pure functional languages.

CHAPTER 5

Towards Slicing

As mentioned in the introduction, this student research project was motivated by Daniel Wasserrab's work on a formally verified and programming language agnostic framework for slicing. Slicing answers the question: Which parts of the program affect a specific statement. Wasserrab mentions several uses for Slicing, such as debugging, testing and verifying software security algorithms.

To instantiate his framework for our functional programming language, we have to transform a program into a control flow graph (CFG) containing semantic information. An instantiation has to provide types for the program state (*state*), for edges (*edge*), nodes (*nodes*), variables (*var*), values (*val*) and definitions for the following functions:

- A predicate *valid-edge* :: *edge* $\Rightarrow$ *bool*, defining the set of edges in the CFG.
- Two functions *source, target* :: *edge* $\Rightarrow$ *node*, indicating the nodes connected by an edge.
- A function *kind* :: *edge* $\Rightarrow$ *state edge-kind* which gives the kind of an edge. An edge is either an updating edge, written $\Uparrow f$ for a function f :: *state* $\Rightarrow$ *state*, or an assertion edge, write $(Q)_{\surd}$ for a predicate Q :: *state* $\Rightarrow$ *bool*.
- A special node *(-Entry-)* indicating the entry node.
- Two sets of variables *Use, Def* :: *node* $\Rightarrow$ *var set* which list, for each node, the variables that are used resp. written to when executing this node.
- A function *state-val* :: *state* $\Rightarrow$ *var* $\Rightarrow$ *val*, obtaining the value of the variable for a given program state. The framework does not actually

work with values, but it uses this function to make statements about changing or unaltered parts of the program state.

These definitions are expected to fulfill certain properties, where edge always refers to a valid edge:

- No edge has the entry node as a target.
- There are no multi-edges, e.g. any edge is uniquely determined by its source and target node.
- The entry edge has empty *Def* and *Use* sets.
- Evaluating an edge does not change the value of any variable not mentioned in its source node's *Def* set.
- If all variables in the *Use* set of a node are equal in two states s and s', then all variables in the *Def* set are equal in the resulting states after evaluating an edge from that node.
- Similarly, if all variables in the *Use* set of a node are equal and an assertion edge from that node would allow traversal in one state, it would also allow traversal in the other state.
- The graph is deterministic, i.e. if two different edges have the same starting node, they are assertion edges whose predicate are mutually exclusive.

Wasserrab optionally introduces special exit nodes, which we skip in this treatment.

5.1 Instantiation

The following documents a plan to instantiate the above definitions for our functional programming language. We have not fully put this into practice.

The first step is to consider the types *node* and *state*. A node together with a state clearly has to carry all information needed to continue evaluating the program. Comparing this with our (standard) semantics functions $\mathcal{F}$ and $\mathcal{C}$,

we see that *node* × *state* have to correspond to *fstate* + *cstate*, the disjoint sum of the arguments vectors:

$$fstate = d \times d\ list \times venv \times contour$$
$$cstate = call \times benv \times venv \times contour$$

The question now is how to distribute these elements on *node* and *state*. Our approach is to put all syntactical information, e.g. expressions already occurring in the syntax tree of the program, into the nodes and the rest into the state. This is a natural choice considering that for imperative programs nodes correspond to statements of the program code.

Clearly, *call* is syntactical. A semantic value of type d is either a closure, a primitive operation, the special value *Stop* or an integer value. Integers values can not occur as the first argument to $\mathcal{F}$. Primitive operations are syntactical and will be put into the node. *Stop* is not a subexpression of the program, but still has a syntactical touch to it, so this will also be put into the node. A closure is a pair *lambda* × *benv*, where the lambda expression is syntactical and the binding environment dynamic. Thus the former goes into the node and the latter into the state. In the end, we reach these definitions:

$$node = lambda + prim + \{Stop\} + call$$
$$state = venv \times d\ list \times benv \times contour$$

These types are actually redundant: For example, a node for a primitive operation does not expect a binding environment. In that case, the arguments will just be looped through unaltered. This actually improves the instantiation, as it keeps the *Def* sets as small as possible.

We also need a special node as the entry node, containing the full program. This leads to Isabelle type definitions printed in Figure 19 on the following page. Note that the type is actually named *synNode*, for reasons explained in the following.

Having thus defined nodes, we turn to the edges of our graph. One would think that these can just be pairs of syntactical nodes, being traversed if the control flow passes from one node to the other, updating the state in doing so. This is not directly possible, as the slicing framework differs between assertion edges and update edges.

We therefore have to split the edge into two, and introduce an intermediate node. Instead of the single edge

$$node1 :: synNode \longrightarrow node2 :: synNode$$

```
types state = venv × d list × benv × contour

datatype synNode
  = StartNode prog
  | StopNode
  | LambdaNode lambda
  | PrimNode prim
  | CallNode call

datatype node
  = SynNode synNode
  | GuardNode synNode synNode
```

```
types edge = node × node

datatype cfgVar =
  Vvar var
 | Vds
 | Vβ
 | Vcnt

datatype cfgVal =
  VLvar contour ⇀ d
 | VLds d list
 | VLβ benv
 | VLcnt contour
```

Figure 19: Isabelle types for the instantiation of the slicing framework

an evaluation step is represented by the two edges

$$SynNode\ node1 \xrightarrow{(Q)\swarrow} GuardNode\ node1\ node2 \xrightarrow{\Uparrow f} SynNode\ node2$$

where the first arrow represents the assertion edge stating "the control flow passes onto *node2*" and the second arrow represents the update edge which actually modifies the state.

The next types to define are the variables and values. To distinguish these from the definitions in the semantics theory, we will call the type for the CFG instantiation *cfgVar* and *cfgVal*. Although not directly obvious from the conditions listed above, the complete *state* has to be accessible via *state-val* to obtain a successful instantiation. Therefore, the set of variables in the graph consists of the set of variables in our program, plus special variables for the argument vector, the binding environment and the contour number in the state vector. The type *cfgVal* has corresponding constructors. Because the variable environment is actually a partial map from variable/contour pairs to semantic values, the value in the CFG of a variable is a partial map from contour to semantic values. The resulting datatype definitions can be seen in Figure 19. The implementation of *state-val* follows immediately.

It remains to define suitable *Def* and *Use* sets. *SynNode*s need to have an empty *Def* set, as they are the origins of assertion edges. A trivially correct choice for the other sets would be *UNIV*::*cfgVar*, the set of all variables. But this would not lead to any useful results. Therefore, the sets have to be cut down as far as possible. For example, the node representing a call expression

$(f\ a_1 \ldots a_n)$ would have a *Def* set of $\{Vds,\ V\beta,\ Vcnt\}$, as it sets the argument vector, pass a contour environment if f happens to evaluate to a closure, and update the contour counter, but does not modify the variable environment. Its *Use* set would encompass $\{V\beta,\ Vcnt\}$ plus variables for each reference among $\{f, a_1, \ldots, a_n\}$.

5.2 A Small Step Semantics

The edges of the CFG are yet to be given their semantic meaning. The model of the CFG as expected by the framework is reminiscent of a small step semantics. Therefore, it would be helpful to have such a semantics. We can easily derive it from the the equations of the denotational semantics, as they are tail recursive, by tracing their arguments of type *fstate* + *cstate* (which corresponds to *synNode* × *state*) as the recursion runs. If we denote with $\langle n : s\rangle \Rightarrow \langle n' : s'\rangle$ the fact that evaluating the node n in state s step to node n' in state s', the rule for call expressions would read:

$$\langle \mathit{CallNode\ (App\ lab\ f\ vs)} : \mathit{(ve,\ ds,\ \beta,\ b)}\rangle$$
$$\Rightarrow \begin{cases} \langle \mathit{LambdaNode\ l} : (\mathit{ve},\ \mathit{ds'},\ \beta',\ \mathit{nb\ b})\rangle, & \text{if } \mathcal{A}\ f\ ve\ \beta = DC\ l\ \beta' \\ \langle \mathit{PrimNode\ p} : (\mathit{ve},\ \mathit{ds'},\ \beta,\ \mathit{nb\ b})\rangle, & \text{if } \mathcal{A}\ f\ ve\ \beta = DP\ p \\ \langle \mathit{StopNode} : (\mathit{ve},\ \mathit{ds'},\ \beta,\ \mathit{nb\ b})\rangle, & \text{if } \mathcal{A}\ f\ ve\ \beta = Stop \end{cases}$$

where $ds'_i = \mathcal{A}\ vs_i\ ve\ \beta$. Note how the argument vector *ds* in the starting state is not used, and how the binding environment stays untouched unless the procedure f evaluates to a closure which carries its own binding environment.

The rule for the special *StartNode* would ignore the given state and set up the initial state:

$$\langle \mathit{ProgNode\ prog} : \mathit{state}\rangle \Rightarrow \langle \mathit{LambdaNode\ l} : (\mathit{empty},\ [\mathit{Stop}],\ \mathit{empty},\ b_0)\rangle$$

and no rule would further evaluate the *StopNode*, which thus becomes the terminal node.

We use this semantics to fill our graph edges with life: The assertion edge from node n towards n' will carry the predicate which is true for a state s if and only if $\langle n : s\rangle \Rightarrow \langle n' : s'\rangle$ for some state s', and the update edge will set the state to just this s'.

It should be possible to prove the equality of the two semantics, i.e.

$$\mathcal{PR}\ prog = i$$
$$\Longleftrightarrow$$
$$\langle StartNode\ prog : (\_, \_, \_, \_)\rangle \Rightarrow^* \langle StopNode : (\_, [i], \_, \_)\rangle$$

where i is the result of the evaluation, $\Rightarrow^*$ the transitive hull of $\Rightarrow$ and $\_$ denotes arbitrary values.

5.3 Connecting to Shivers

If that could be done, a similar proof will relate the small step semantics with the exact nonstandard semantics: For each step in the series $\langle n : s\rangle \Rightarrow \langle n' : s'\rangle$, if n' is not a *CallNode*, we find a corresponding entry in the call cache returned by the nonstandard semantics function $\mathcal{PR}$. Call caches are of the type $(label \times benv \rightharpoonup d)$. The label identifies a subexpression of the program, and thus a *synNode*. The binding environment is ignored for this purpose. The semantic value d contains a procedure and hence can be identified with a *synNode*.

We have shown that the call cache returned by $\widehat{\mathcal{PR}}$ is a safe approximation to the exact call cache. Therefore, we will find corresponding entries there as well. By the converse argument we see that it suffices to feed such edges to the slicing framework that we can statically derive using Shivers' abstract nonstandard semantics. This will make the constructed graph considerably smaller and thus more exact, and we hope this allows for a more powerful analysis.

5.4 Hopes and Fears

The result of a slicing analysis is, given a node that we are interested in, a set of nodes that do not affect our node. In an imperative language, this would allow us to replace the statements which correspond to these nodes by no-op statements without affecting the statement which corresponds to our interesting node.

In our functional, continuation-passing style language, it is not so clear how to interpret the result of the analysis. Clearly, we cannot just replace such nodes

with no-ops, as that would halt evaluation altogether. It might be possible to omit the variable bindings or argument vector setting in such nodes and then, in a post-processing step, collapse call-lambda-pairs where no arguments are passed and no variables are bound.

A problem in the implementation we suggested in this chapter could be caused by our treatment of the contour counter, represented by the graph variable *Vcnt*. Every node reads it, and almost all nodes increase it. This could cause a lot of unwanted dependencies between the nodes, rendering the slicing analysis worthless.

For a possible remedy additional nodes could be inserted in the graph whose only effect is to increase the contour counter and thus removing *Vcnt* from the other nodes' *Def* sets. We expect that these contour nodes are never removed by slicing, and that these would only depend on other contour nodes plus potentially nodes which represent conditional statements.

A similar problem is posed by the binding environment in the graph variable $V\beta$, which is also written by each lambda expression and used by each call expression. Again, dedicated nodes could avoid having $V\beta$ in the *Def* sets of lambda expressions.

The combination of a call and a lambda expression is roughly equivalent to an assignment statement in an imperative language. Combining these in one step, we can remove the argument vector variable *Vds* from our scheme and split the assignment of individual arguments into separate nodes. This would avoid unnecessary dependencies between the loop counter and the loop accumulator in a loop implemented by a recursive function.

Splitting the binding environment into a set of graph variables, one per binding position, should be considered. It is not clear whether this will improve the representation without further modifications, as the evaluation of a lambda expression to a closure bundles the complete binding environment, which requires all of these graph variables to appear in the *Use* and *Def* set of a call.

In Section 2.6 the problems of the 0CFA-abstraction in the presence of higher order functions such as $\circ$ are mentioned. These also occur here and would lead to unwanted edges in the control graph. A transformation that allows to use the additional information provided by 1CFA is tricky, though, as binding environments do not occur in the static part of the graph. A possible solution is to duplicate nodes, once per context identified by the 1CFA-abstraction, at the cost of an increased graph.

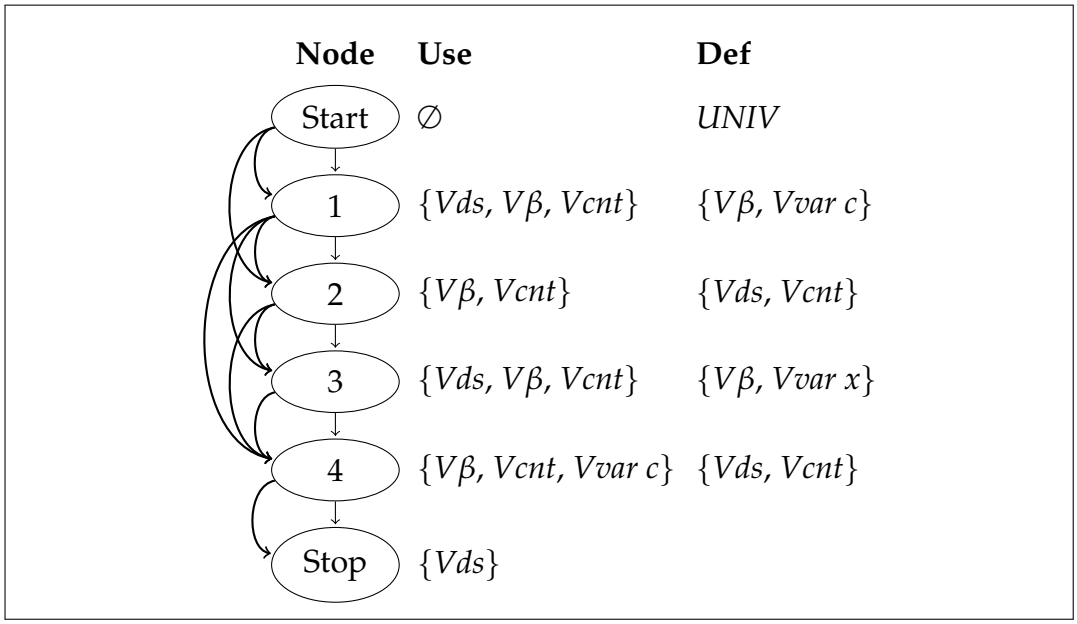

Figure 20: Naïve instantiation

5.5 Example

The smallest example of a program with obviously useless instructions would be the following code:

```
(λ c. (λx. c 1) 0).
 1    2 3   4
```

Figure 20 contains the naive call graph without any of the modifications in the previous section. The program's control flow is purely linear, denoted by down-pointing edges. Edges in the other direction denote dependencies between the nodes. The intermediate nodes which we added per edge to fulfill the framework's requirements are not shown.

In this form, every node influences the final node and we do not obtain any useful information from the slicing analysis. In Figure 21, we improved the instantiation by introducing special nodes to update the contour counter *Vcnt* resp. the contour environment $V\beta$. The figure contains only the dependency arrows relevant in the analysis of the stop node.

In this case, the analysis provides a useful result: The nodes 2 and 3, representing the useless call and the useless lambda, do not influence the Stop node. The corresponding extra nodes are still in the set of depending nodes. In that sense, they form the skeleton of the program flow and are never removed by slicing.

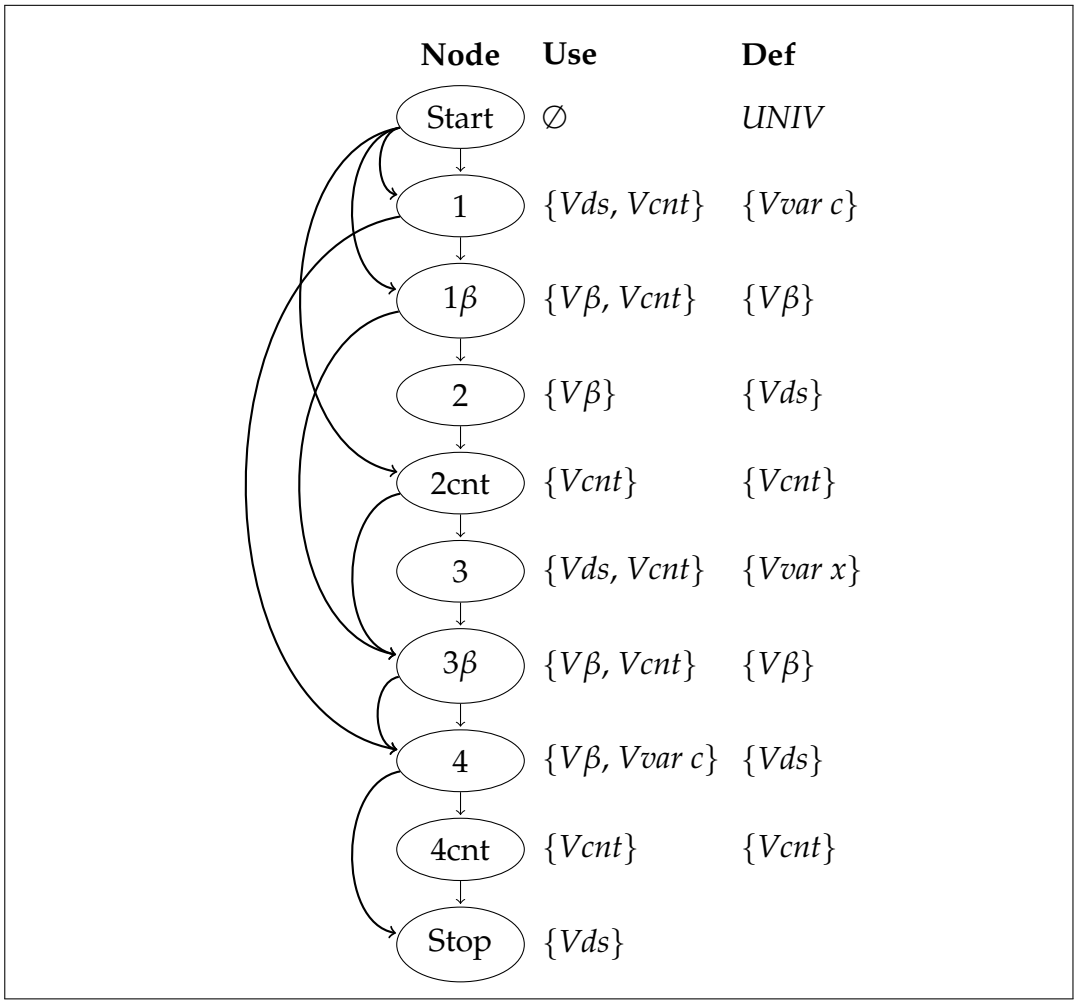

Figure 21: Improved instantiation

CHAPTER 6

Conclusion

We have successfully formalized Olin Shivers' control flow analysis for functional programming languages and proven it correct. We employed the theorem prover Isabelle and chose the logic package *HOLCF* to base our work on. This turned out to be the right choice for the implementation of denotational semantics.

The formalization was, for the greatest part, straight forward. Where it was not, it was still possible to obtain the desired results, after some experimentation. Altogether, we were satisfied with the Isabelle system.

We outlined a possible connection to Daniel Wasserrab's framework for formally verified slicing, discussed its problems and possible remedies. By a simple example, we showed that such a slicing analysis can indeed be useful when applied to functional programs.

It would be interesting to put this plan into practice and implement an instantiation of the slicing framework for functional languages. One of the large open questions here is how to interpret the results obtained by the slicing analysis.

Bibliography

[1] *Graphviz – DOT and DOTTY*, http://www.graphviz.org/, Version 2.26.3.

[2] *Isabelle: A Generic Theorem Proving Environment*, http://isabelle.in.tum.de/, Version 2009-2.

[3] Christophe Favergeon, *HPDF: Generation of PDF documents*, http://hackage.haskell.org/package/HPDF-1.4.2, February 2009.

[4] Holger Gast, *i3p*, http://www-pu.informatik.uni-tuebingen.de/i3p/, Version 1.0.8.

[5] John Hughes, *The design of a pretty-printing library*, Advanced Functional Programming (Johan Jeuring and Erik Meijer, eds.), Lecture Notes in Computer Science, vol. 925, Springer Berlin / Heidelberg, 1995, pp. 53–96.

[6] Gerwin Klein and Tobias Nipkow, *A machine-checked model for a Java-like language, virtual machine and compiler*, **28** (2006), no. 4, 619–695.

[7] Alexander Krauss, *Defining recursive functions in Isabelle/HOL*, http://isabelle.in.tum.de/doc/functions.pdf.

[8] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oscar Slotosch, *HOLCF = HOL + LCF*, Journal of Functional Programming **9** (1999), 191–223.

[9] Olin Shivers, *Control-flow analysis of higher-order languages*, Ph.D. thesis, 1991.

[10] Sid Steward, *pdftk – the pdf toolkit*, http://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/, Version 1.41.

[11] Daniel Wasserrab, *From formal semantics to verified slicing - a modular framework with applications in language based security*, Ph.D. thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, October 2010.

Erklärung

HIERMIT erkläre ich, Joachim Breitner, dass ich die vorliegende Studienarbeit selbständig und ausschließlich unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe.

Ort, Datum

Unterschrift