Declarative semantics for functional languages: compositional, extensional, and elementary

JEREMY G. SIEK, Indiana University

We present a semantics for an applied call-by-value λ-calculus that is compositional, extensional, and elementary. We present four different views of the semantics: 1) as a relational (big-step) semantics that is not operational but instead declarative, 2) as a denotational semantics that does not use domain theory, 3) as a non-deterministic interpreter, and 4) as a variant of the intersection type systems of the Torino group. We prove that the semantics is correct by showing that it is sound and complete with respect to operational semantics on programs and that is sound with respect to contextual equivalence. We have not yet investigated whether it is fully abstract. We demonstrate that this approach to semantics is useful with three case studies. First, we use the semantics to prove correctness of a compiler optimization that inlines function application. Second, we adapt the semantics to the polymorphic λ-calculus extended with general recursion and prove semantic type soundness. Third, we adapt the semantics to the call-by-value λ-calculus with mutable references. All of the definitions and proofs in this paper are mechanized in Isabelle in under 3,000 lines.

CCS Concepts: •**Software and its engineering → Functional languages; Semantics;**

Additional Key Words and Phrases: declarative, denotational, relational, lambda calculus, intersection types

ACM Reference format:
Jeremy G. Siek. 2017. Declarative semantics for functional languages: compositional, extensional, and elementary. *Proc. ACM Program. Lang.* 1, 1, Article 1 (January 2017), 29 pages.
DOI: X

1 INTRODUCTION

When I think of the number of headaches I have caused people in Computer Science who have tried to figure out the mathematical details of the Theory of Domains, I have to cringe. —Scott (1982)

A realisation struck me around then... operational semantics was both simple and given by elementary mathematical means. So why not consider dropping denotational semantics and, once again, take operational semantics seriously? — Plotkin (2004)

This paper describes a semantics for an applied call-by-value (CBV) λ-calculus that satisfies three important properties: it is *compositional*, *extensional*, and *elementary*.

compositional It is defined by structural recursion on the syntax of programs (Winskel 1993).

extensional It only defines behavior that is externally observable (Schmidt 2003).

elementary It only uses basic discrete mathematics.

The two predominant approaches to semantics do not satisfy all three properties. Denotational semantics is compositional and extensional but not elementary whereas operational semantics is elementary but neither compositional nor extensional.

Compositionality enables proof by structural induction on the syntax, which simplifies proofs of properties such as type soundness and implementation correctness. Extensionality is beneficial

2017. XXXX-XXXX/2017/1-ART1 $X
DOI: X

because, ultimately, a language specification must be extensional; intensional semantics require the circuitous step of erasing internal behavior. The use of only elementary mathematics is beneficial because language specifications are a form of communication between computer scientists and because the size and difficulty of a language's metatheory depends on the complexity of the mathematics. It is not obvious how to construct a semantics that is compositional, extensional, and elementary, especially with the higher-order features that appear in modern languages. In this paper we present such a semantics and demonstrate that the approach can handle first-class functions in an untyped setting, first-class parametric polymorphism in a typed setting, and even mutable references in an imperative setting.

Historically, the predominant approaches to specifying programming languages have been operational semantics (Church 1932; Felleisen et al. 1987; Landin 1964; McCarthy 1960; Plotkin 1981) and denotational semantics (Scott 1970; Scott and Strachey 1971). Operational semantics are elementary but neither extensional nor compositional. Small-step operational semantics are intensional in that the input-output behavior of a program is a by-product of a sequence of transitions. Big-step operational semantics are intensional in that the value of a λ abstraction is a syntactic object, a closure (Kahn 1987). The lack of compositionality in operational semantics imposes significant costs when reasoning about programs: sophisticated techniques such as logical relations (Plotkin 1973; Statman 1985) and simulations (Aczel 1988; Milner 1995) are often necessary. For example, the correctness proofs for the CompCert C compiler made extensive use of simulations but sometimes resorted to translation validation in cases where verification was too difficult or expensive (Leroy 2009; Tristan and Leroy 2008). Likewise, the logical relations necessary to handle modern languages are daunting in their complexity (Ahmed et al. 2009; Hur and Dreyer 2011).

Denotational semantics are compositional and extensional but they are not elementary. The increase in mathematical complexity required by the denotational approaches is striking when comparing mechanized metatheory (Aydemir et al. 2005). For example, the mechanization of the λ-calculus based on denotational semantics by Benton et al. (2009) is 11,000 LOC and by Dockins (2014) is 54,104 LOC (though it is difficult to see how much of that is strictly necessary for defining the semantics). In contrast, a mechanized definition of the λ-calculus using operational techniques is under 100 LOC. This difference in complexity justifies the prevailing attitude that operational semantics are better for language specification, even though they are neither compositional nor extensional.

But what if there was a compositional semantics that was also elementary? What if its definition could be mechanized in under 100 LOC? We present such a semantics in this paper. It achieves compositionality by being declarative, that is, by strategic use of the existential quantifier.[1] Section 2 describes four ways to view this semantics:

(1) as a relational (aka. big-step) semantics, i.e., a relation on programs and output values,
(2) as a denotational semantics, i.e., a function from programs to their denotations,
(3) as a non-deterministic interpreter, and
(4) as a type system with intersection types.

In fact, there is a fifth way to view this semantics, as a domain logic (Abramsky 1987), which is closely related to the type system view. The denotational view of the declarative semantics is unusual in that it does not use CPOs, limits, or any sophisticated notions from domain theory.

The main insight of the declarative semantics is so simple it is hard to believe: in the context of a program, the meaning of a λ abstraction is *some* finite graph (i.e. lookup table) whose domain

[1]The same is true for the declarative semantics of logic programming languages (Emden and Kowalski 1976; Lloyd 1984).

happens to be large enough to handle all later uses of the λ in the execution of the program. This too-simple idea needs one adjustment to make it work. Given an argument v, a naive approach to table lookup would find an entry (v_1, v_2) such that $v = v_1$ and then return v_2. However, in the case when v and v_1 are themselves functions, i.e., tables, testing for equality is overly restrictive. Instead the semantics allows *subsumption* when comparing v to v_1, that is, v matches v_1 when v contains every entry in v_1, that is, when $v_1 \subseteq v$. These insights have their origins in research on intersection types (Coppo et al. 1979) and filter models of the λ-calculus (Barendregt et al. 1983; Plotkin 1993). Also, our domain of values corresponds to D^0 in the work on semantic subtyping where it is used to model types but not programs (Frisch et al. 2008). The relational, denotational, and interpreter versions of this semantics are new artifacts. The type system with intersection types presented here is a new variation on the type systems studied by the Torino group and their colleagues (Alessi et al. 2006; Barendregt et al. 2013; Coppo et al. 1979; Hindley 1982). The mechanized definition of all three semantics is under 150 LOC in the Isabelle proof assistant (Nipkow et al. 2007). The goals of this paper are to advertise these ideas and make them accessible to a wider audience, mechanize the correctness of this approach, and to explore applications outside of pure λ-calculi.

There are several relevant notions of correctness that we address in the setting of an applied call-by-value untyped λ-calculus. Is the declarative semantics sound and complete with respect to standard operational semantics for programs? Let e be a program and n an integer. Let $\mathcal{E}$ be the denotation function for our semantics (Section 2.2) which maps an expression and environment to a set of values, e.g., $\mathcal{E}[\![n]\!]\emptyset = \{n\}$. We write $e \Downarrow n$ to say that e evaluates to n under the standard operational semantics. Then we have the following results.

Theorem 1 (Sound wrt. op. sem.). *(aka. Adequacy) If $\mathcal{E}[\![e]\!]\emptyset = \mathcal{E}[\![n]\!]\emptyset$, then $e \Downarrow n$.*

Theorem 2 (Complete wrt. op. sem.). *If $e \Downarrow n$, then $\mathcal{E}[\![e]\!]\emptyset = \mathcal{E}[\![n]\!]\emptyset$.*

We prove Theorem 1 using a logical relation (Section 3.1). We prove Theorem 2 by adapting techniques from intersection types and the pure λ-calculus (Alessi et al. 2003) (Section 3.2). Both proofs are mechanized in Isabelle. They share 100 LOC of lemmas and then the proof of the first theorem is 222 LOC and the second is 317 LOC.

Another notion of correctness is needed to justify using the semantics in correctness proofs for compiler optimizations: soundness with respect to contextual equivalence (Morris 1968). For two arbitrary expressions e and e' we prove that if they are denotationally equal, then they are contextually equivalent. Note that $\mathcal{E}$ is curried and only partially applied in the theorem below. Also, $e \Downarrow$ means that e terminates according to the operational semantics.

Theorem 3 (Sound wrt. Contextual Equivalence).
If $\mathcal{E}[\![e]\!] = \mathcal{E}[\![e']\!]$, then $C[e] \Downarrow$ iff $C[e'] \Downarrow$ for any closing context C.

The proof strategy is to show that $\mathcal{E}$ is a congruence and then to apply soundness and completeness for programs (Gunter 1992) (Section 3.3). This proof is mechanized in Isabelle in 172 LOC. The other direction, that contextual equivalence implies denotational equivalence, would give us *full abstraction* (Abramsky et al. 2000; Hyland and Ong 2000), but we have not investigated that direction. We note that Ronchi Della Rocca and Paolini (2004) proved full abstraction for a related filter model.

Next we investigate whether the declarative approach to semantics is useful for practical applications. This is a much larger question and will require many case studies to answer in full. Here we take three steps down that road. Our first case study applies our semantics in the proof of correctness for a compiler optimization, in particular, function inlining (Section 4). We prove, in Isabelle, that the output of the compiler is denotationally equivalent to the input, so it preserves contextual equivalence by Theorem 3. The definition of the optimizer and the proof of correctness are just 56 LOC.

Our second case study adapts the declarative semantics to a statically-typed language with parametric polymorphism: System F with general recursive functions. This is particularly interesting because parametric polymorphism was a point of difficulty for denotational semantics (Breazu-Tannen and Coquand 1988; Bruce et al. 1990; Coquand et al. 1989; McCracken 1979; Pitts 1987). For example, the original set-theoretic semantics were incorrect (Reynolds 1984). We demonstrate how to mechanize a proof of type soundness using the declarative semantics. In particular, we prove that well-typed programs cannot go wrong (Milner 1978) (Section 5). Our proof, unlike Milner's proof for Mini ML, defines a meaning for universal types and type variables and is therefore more reminiscent of proofs of Parametricity. Indeed, the key lemma is a Compositionality Lemma (Skorstengaard 2015). The mechanization of our second case study is 510 LOC.

Our third case study gives a semantics to a simply-typed λ-calculus extended with mutable references (Section 6). This demonstrates that our declarative approach scales to imperative languages, i.e., languages with a heap. Surprisingly, our domain of values does not need to change in a significant way because we represent the heap the same way we represent functions, as a finite table. We simply add a new kind of value for addresses (to distinguish them from integers). The definition is 153 LOC.

Contributions. To summarize the above, the technical contributions of this paper are as follows.

(1) A declarative semantics for an applied CBV λ-calculus that is compositional, extensional, and elementary. We present four views of the semantics: a relational semantics, a denotational semantics, a non-deterministic interpreter, and an intersection type system (Section 2).
(2) Correctness proofs: the semantics is sound and complete with respect to operational semantics, and is sound with respect to contextual equivalence (Section 3).
(3) Case studies that begin to demonstrate the usefulness of our declarative semantics:
    - A proof of correctness for a compiler optimization pass that performs inlining (Section 4).
    - A proof of semantic type soundness for System F with general recursion (Section 5).
    - A semantics for a CBV λ-calculus with mutable references (Section 6).

All of the results in this paper are mechanized in Isabelle and are in the supplementary material.

2 DECLARATIVE SEMANTICS OF THE CBV λ-CALCULUS

But f is a function*; an infinite object. What does it mean to "compute" with an "finite" argument? In this case it means most simply that $h(f)$ is determined by asking of f finitely many questions: $f(m_0), f(m_1), ..., f(m_{k-1})$.* —Scott (1993)

We first present the declarative semantics in the context of an applied CBV untyped λ-calculus. The syntax of this language is defined by the grammar in Figure 1. We write n for integers, $e_1 \oplus e_2$ for arithmetic operations, x for variables, $\lambda x.\, e$ for abstraction, $e_1\ e_2$ for application, and **if** e_1 **then** e_2 **else** e_3 for conditionals.

As alluded to in the introduction, the main insight of the new semantics is to represent λ's with a table. So we inductively define *values* as follows.

$$\begin{array}{rcll} t & ::= & \{(v_1, v_1'), \ldots, (v_n, v_n')\} & \text{tables} \\ v \in \mathbb{V} & ::= & n \mid t & \text{values} \end{array}$$

$$\begin{array}{rcll}
x & \in & \mathbb{X} & \text{variables} \\
n & \in & \mathbb{Z} & \text{integers} \\
\oplus & \in & \{+, \times, -, \ldots\} & \text{arithmetic operators} \\
e \in \mathbb{E} & ::= & n \mid e \oplus e \mid x \mid \lambda x.\, e \mid e\ e \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e & \text{expressions}
\end{array}$$

Fig. 1. Syntax of a call-by-value λ calculus with integer arithmetic.

We write $\mathrm{fset}(\alpha)$ for all the finite sets with element drawn from α. For readers who prefer domain equations, values are the least solution to the following equation.

$$\mathbb{V} = \mathbb{Z} + \mathrm{fset}(\mathbb{V} \times \mathbb{V})$$

In Isabelle, we use a `datatype` definition to define values. We have experimented with representing tables as a `list` of pairs or as a finite set (`fset`) of pairs and found that the later is much better because then the builtin equality on values has the correct meaning.

As usual, a runtime *environment* $\rho \in \mathbb{X} \rightharpoonup \mathbb{V}$ is a partial function from variables to values. We write $\emptyset$ for the nowhere-defined function and $\rho(x{:=}v)$ for the function that maps x to v and any other variable y to $\rho(y)$.

2.1 The Declarative Semantics as a Relational Semantics

For readers familiar with operational semantics, the declarative semantics is perhaps easiest to understand as a relational semantics (aka. big-step or natural semantics (Kahn 1987)). But unlike most other relational semantics for the λ-calculus, this semantics is not operational, i.e., it is not suggestive of an implementation. We write $\rho \vdash e \Rightarrow v$ for the relation that holds when e evaluates to v in environment ρ. For example,

$$\emptyset \vdash (20 + 1) \times 2 \Rightarrow 42 \quad \text{and} \quad \neg\ \emptyset \vdash 21 \times (2 + 3) \Rightarrow 42.$$

We represent a function by a finite table, which means that the user of the semantics can ask whether a function maps certain inputs to certain outputs. For example, we have

$$\emptyset \vdash \lambda x.\, x{+}1 \Rightarrow \{(41, 42), (-10, -9)\},\ \emptyset \vdash \lambda x.\, x{+}1 \Rightarrow \{(1, 2), (41, 42)\},\ \text{and}\ \neg\ \emptyset \vdash \lambda x.\, x{+}1 \Rightarrow \{(1, 1)\}.$$

For deterministic programming languages, a relational semantics typically specifies a unique output value for a program. As you can see here, our declarative semantics instead relates a λ abstraction to many different tables. The tables should be interpreted as providing only positive information and not negative, that is, the absence of an entry in a table does not mean anything.

For programs whose output is a function, as in the above example, it helps to think of the parameter of the function as a query to the user of the semantics. In the above, the semantics asks for a value of x and gets several different responses from the user (41, -10, and 1). So the query-response nature of the semantics gets flipped when dealing with functions.

One of the main features of the λ-calculus is that functions are first class and can therefore be passed to other functions. For example, if we let $t_0 = \{(1, 2)\}$ and $t_1 = \{(0, 1), (1, 2)\}$, then we have

$$\emptyset \vdash \lambda x.\, x \Rightarrow \{(t_0, t_0), (t_1, t_1)\} \ \text{ and } \ \neg\ \emptyset \vdash \lambda x.\, x \Rightarrow \{(t_0, t_1)\}.$$

The way in which this program can return a function brings up an interesting question. What if instead of asking whether, given input t_1, the output is a function that maps 0 to 1 and 1 to 2, we only ask if the output maps 1 to 2.

$$\emptyset \vdash \lambda x.\, x \Rightarrow \{(t_1, t_0)\}$$

One would think that the answer should still be “yes”. It turns out the answer had better be yes! Otherwise the semantics cannot handle self application, which would be a deal breaker for the λ-calculus. Consider the following example of self application. Is there a table t_3 such that we get 1 as a response.

$$\emptyset \vdash \lambda f.\, f\ f \Rightarrow \{(t_3, 1)\})$$

It seems that the application of f to itself would require that $(t_3, 1) \in t_3$, but such circularity is not possible because we have inductively defined our values[2]. But if we allow a “yes” response when the query is a subset of the answer, then for example, we could let $t_3 = \{(\emptyset, 1)\}$ and then answering t_3 for the query $\emptyset$ is fine because $\emptyset \subseteq t_3$. This can be thought of as a kind of subsumption: one can use a more-defined table in places where a less-defined table is expected.

To make our definitions uniform, we lift the subset operation to all values, writing $v \sqsubseteq v'$, which we define as follows:

$$n \sqsubseteq n \qquad \frac{t \subseteq t'}{t \sqsubseteq t'}$$

Proposition 2.1 ($\sqsubseteq$ is a partial order).

(1) *$v \sqsubseteq v$ (reflexive) and*
(2) *if $v_1 \sqsubseteq v_2$ and $v_2 \sqsubseteq v_3$, then $v_1 \sqsubseteq v_3$ (transitive).*
(3) *if $v_1 \sqsubseteq v_2$ and $v_2 \sqsubseteq v_1$, then $v_1 = v_2$ (antisymmetric).*

Motivated by the above discussion, we give an inductive definition of the relation $\rho \vdash e \Rightarrow v$ in Figure 2. The rules for λ abstraction and application are the most important and quite different from prior relational semantics. The rule for $\lambda x.\, e$ states that the result is some table t if, for every input-output pair (v, v') in t, extending the environment with $x{:=}v$ and evaluating the body e results in v'. We like to think of the table t as being produced by an oracle that looks into the future and chooses a table whose domain is big enough to handle all of the subsequent uses of it in the program execution. The premise of the rule makes sure that the table t is a subset of the function described by $\lambda x.\, e$. The rule for a function application $(e_1\ e_2)$ performs table lookup. Expression e_1 evaluates to a table t and e_2 evaluates to v_2. The rule then finds a match for the argument v_2 in the table t and obtains the associated output value. The twist is that when finding a match, instead of using equality on values we use $\sqsubseteq$ to allow for the subsumption that we discussed above. Also, we use $\sqsubseteq$ a second time to relate the output value from the table and the result value v, to make sure that subsumption is admissible (Proposition 2.3). The rules for integers, arithmetic, and the conditional **if** expression are straightforward. Regarding variables, the twist is that a variable evaluates to any value v such that $v \sqsubseteq \rho(x)$, again to make sure that subsumption is admissible.

Example of Recursion. In our earlier discussion we showed how the declarative semantics can support self application, but the reader may yet be wondering whether it supports recursion. We ultimately answer the question in Section 3, by proving that the declarative semantics is equivalent to the operational semantics. Nevertheless, it may be helpful to see an example of a recursive function to see how it works. The main idea is that of a Matryoshka doll: we shall nest ever-smaller versions of a recursive function inside of itself.

Recall the Z combinator (the version of the Y combinator for strict languages):

$$M \equiv \lambda x.\, f\ (\lambda v.\, (x\ x)\ v) \qquad Z \equiv \lambda f.\, M\ M$$

The factorial function is defined as follows, with a parameter r for calling itself recusively. We give names (F and H) to the λ abstractions because we shall define tables for each of them.

$$F \equiv \lambda n.\, \textbf{if}\, n = 0\, \textbf{then}\, 1\, \textbf{else}\, n \times r\ (n-1) \qquad H \equiv \lambda r.\, F \qquad \mathit{fact} \equiv Z\ H$$

[2]Could we instead coinductively define the values?

$$\boxed{\rho \vdash e \Rightarrow v}$$

$$\frac{}{\rho \vdash n \Rightarrow n} \qquad \frac{\rho \vdash e_1 \Rightarrow n_1 \quad \rho \vdash e_2 \Rightarrow n_2}{\rho \vdash e_1 \oplus e_2 \Rightarrow n_1 \oplus n_2} \qquad \frac{v \sqsubseteq \rho(x)}{\rho \vdash x \Rightarrow v}$$

$$\frac{\forall (v, v') {\in} t.\ \rho(x{:=}v) \vdash e \Rightarrow v'}{\rho \vdash \lambda x.\, e \Rightarrow t} \qquad \frac{\begin{array}{cc} \rho \vdash e_1 \Rightarrow t & \rho \vdash e_2 \Rightarrow v_2 \\ (v_3, v_3') \in t \quad v_3 \sqsubseteq v_2 & v \sqsubseteq v_3' \end{array}}{\rho \vdash (e_1\ e_2) \Rightarrow v}$$

$$\frac{\rho \vdash e_1 \Rightarrow n \quad n \neq 0 \quad \rho \vdash e_2 \Rightarrow v}{\rho \vdash \textbf{if}\, e_1\, \textbf{then}\, e_2\, \textbf{else}\, e_3 \Rightarrow v} \qquad \frac{\rho \vdash e_1 \Rightarrow 0 \quad \rho \vdash e_3 \Rightarrow v}{\rho \vdash \textbf{if}\, e_1\, \textbf{then}\, e_2\, \textbf{else}\, e_3 \Rightarrow v}$$

Fig. 2. The declarative semantics for CBV λ-calculus as a relational semantics.

We begin with the tables for F: we define a function F_t that gives the table for just one input n; it simply maps n to n factorial.

$$F_t(n) = \{(n, n!)\}$$

The tables for H map a factorial table for $n - 1$ to a table for n. We invite the reader to check that $\emptyset \vdash H \Rightarrow H_t(n)$ for any n.

$$H_t(n) = \{(\emptyset, F_t(0)), (F_t(0), F_t(1)), \ldots, (F_t(n-1), F_t(n))\}$$

Next we come to the most important part: describing the tables for M. Recall that M is applied to itself; this is where the analogy to Matryoshka dolls comes in. We define M_t by recursion on n. Each $M_t(n)$ extends the smaller version of itself, $M_t(n-1)$, with one more entry that maps the smaller version of itself to the table for factorial of $n - 1$.

$$\begin{aligned} M_t(0) &= \emptyset \\ M_t(n) &= M_t(n-1) \cup \{(M_t(n-1), F_t(n-1))\} \end{aligned}$$

The tables M_t are valid meanings for M because $f{:=}H_t(k) \vdash M \Rightarrow M_t(n+1)$ for any $n \leq k$. The application of M to itself is OK, that is, $f{:=}H_t(k) \vdash M\ M \Rightarrow F_t(n)$, because we have $(M_t(n), F_t(n)) \in M_t(n+1)$, $M_t(n) \sqsubseteq M_t(n+1)$, and $F_t(n) \sqsubseteq F_t(n)$.

To finish things up, the tables for Z map the H_t's to the factorial tables.

$$Z_t(n) = \{(H_t(n), F_t(n))\}$$

Then we have $\emptyset \vdash Z \Rightarrow Z_t(n)$ for all n. So $\emptyset \vdash Z\ H \Rightarrow F_t(n)$ for any n and therefore $\emptyset \vdash \mathit{fact}\, n \Rightarrow n!$.

Tables as Relations vs. Functions. We have defined function tables as finite relations and not finite partial functions. We conjecture that using functions would also work, but it would significantly complicate the join operator that plays an important role in the proof of completeness wrt. operational semantics (Section 3.2). Using relations, the join operator is simply set union. To see how choosing using relations impacts the semantics, consider the program $\lambda y.\, \lambda x.\, x$ that ignores its first argument and returns the identity function. As we have seen above, there are many different tables in the meaning of $\lambda x.\, x$. For example, $\emptyset$, $\{(0,0)\}$, and $\{(8,8),(5,5)\}$. So the above program can produce tables that map the same input, say 1, to several different outputs.

$$\emptyset \vdash \lambda y.\, \lambda x.\, x \Rightarrow \{(1, \emptyset),\ (1, \{(0,0)\}),\ (1, \{(8,8),(5,5)\})\}$$

We conjecture that the tables in the meaning of an expression have the property of mapping consistent inputs to consistent outputs, for some appropriate definition of consistency.

Evaluation. With the definition of Figure 2 in hand, we can evaluate it with respect to our earlier claims. First, note that it is compositional. Each rule in the definition uses the evaluation relation recursively on the immediate sub-expressions. The semantics is extensional in that it maps programs to their outputs without the need to mention internal behavior. The semantics is elementary in that it requires only first-order logic, sets, and inductive definitions. The semantics is declarative in that it makes non-trivial use of existential quantifiers, although this is difficult to see because the existentials are implicit in rule-based presentations. Krishnaswami (2013) defines "declarative" in this context to mean that the relation cannot be well-moded in the sense of a logic program. For example, suppose you choose to treat ρ and e as inputs and v as the output. Then the rule for λ abstraction is not well-moded because the table t is not an output in the premise. Next suppose you choose ρ, e, and v as inputs. Then the rule for function application is not well-moded because t and v_2 do not occur in output positions, only input (not to mention v_3 and v_3').

Comparison to Operational Big-Step Semantics. Let us compare Figure 2 to the standard relational semantics (Kahn 1987), for which we write $\varrho \vdash_s e \Rightarrow w$. ($w$ ranges over values and ϱ ranges over environments of the standard relational semantics.) The difference between the two semantics lies in the function values. In the standard semantics, a function value is a closure of the form $\langle \lambda x.\, e, \varrho \rangle$.

$$w \in \mathbb{W} ::= n \mid \langle \lambda x.\, e, \varrho \rangle$$

Therefore the rule for application is not compositional: in the third premise it is recursive in the body of the λ, which is not an immediate sub-expression of the application expression.

$$\frac{}{\varrho \vdash_s \lambda x.\, e \Rightarrow \langle \lambda x.\, e, \varrho \rangle} \qquad \frac{\varrho \vdash_s e_1 \Rightarrow \langle \lambda x.\, e, \varrho' \rangle \quad \varrho \vdash_s e_2 \Rightarrow w_2 \quad \varrho'(x := w_2) \vdash_s e \Rightarrow w}{\varrho \vdash_s e_1\ e_2 \Rightarrow w}$$

On the other hand, the standard rules for λ abstraction and application are well-moded and therefore suggestive of an implementation. The standard relational semantics is extensional when the output of the program is an integer, but not when the output is a function because a closure is an intensional representation of a function.

Both the standard relational semantics and our declarative version do not distinguish between programs that encounter runtime type errors and programs that diverge. There are two common approaches to making this distinction: use step-indexing (Amin and Rompf 2017; Owens et al. 2016; Siek 2012) or make errors explicit (Milner 1978). We do not yet know how to apply step-indexing to the declarative semantics. We make use of explicit errors in Sections 5 and 6.

2.2 The Declarative Semantics as a Denotational Semantics

For readers familiar with denotational semantics, the declarative semantics is easiest to understand as a function from a program to its output. We can obtain such a function from the relational semantics by exploiting the usual isomorphism between relations and functions into sets. In this particular case, we have the following isomorphism.

$$(\mathbb{E} \times (\mathbb{X} \rightharpoonup \mathbb{V}) \times \mathbb{V}) \to \mathsf{Bool} \quad \cong \quad \mathbb{E} \to (\mathbb{X} \rightharpoonup \mathbb{V}) \to \mathsf{set}(\mathbb{V})$$

So the domain for our denotational semantics is set($\mathbb{V}$). We define function $\mathcal{E}$ in Figure 3 to take an expression and an environment and return a set of values. Its definition is derived from the relational semantics (Figure 2) in the following way. Each case of the function is defined using a set comprehension, with the output value from the conclusion of the rule in Figure 2 used to the left of the bar and the premises of the rule used to the right of the bar. As usual, we enclose expressions in semantic brackets, as in $[\![e]\!]$, to help distinguish them from the meta-level notation.

$$\boxed{\mathcal{E}[\![e]\!]\rho}$$

$$\begin{aligned}
\mathcal{E}[\![n]\!]\rho &= \{n\}\\
\mathcal{E}[\![e_1 \oplus e_2]\!]\rho &= \{n_1 \oplus n_2 \mid \exists n_1 n_2.\, n_1 \in \mathcal{E}[\![e_1]\!]\rho \land n_2 \in \mathcal{E}[\![e_2]\!]\rho\}\\
\mathcal{E}[\![x]\!]\rho &= \{v \mid v \sqsubseteq \rho(x)\}\\
\mathcal{E}[\![\lambda x.\, e]\!]\rho &= \{t \mid \forall (v, v') \in t.\, v' \in \mathcal{E}[\![e]\!]\rho(x{:=}v)\}\\
\mathcal{E}[\![e_1\ e_2]\!]\rho &= \left\{ v \;\middle|\; \begin{array}{l} \exists t v_2 v_3 v_3'.\, t{\in}\mathcal{E}[\![e_1]\!]\rho \land v_2{\in}\mathcal{E}[\![e_2]\!]\rho \\ \land\, (v_3, v_3') \in t \land v_3 \sqsubseteq v_2 \land v \sqsubseteq v_3' \end{array} \right\}\\
\mathcal{E}[\![\textbf{if}\, e_1\, \textbf{then}\, e_2\, \textbf{else}\, e_3]\!]\rho &= \left\{ v \;\middle|\; \exists n.\, n \in \mathcal{E}[\![e_1]\!]\rho \; \begin{array}{l} \land\, (n \neq 0 \implies v \in \mathcal{E}[\![e_2]\!]\rho) \\ \land\, (n = 0 \implies v \in \mathcal{E}[\![e_3]\!]\rho) \end{array} \right\}
\end{aligned}$$

Fig. 3. The declarative semantics for CBV λ-calculus as a denotational semantics.

This denotational semantics is similar to the $P\omega$ model of Scott (1976) in that the domain is sets of values. The primary difference is that in $P\omega$ the domain uses the space of continuous functions, which handles functions with infinite graphs. In contrast, our semantics stratifies the domain, separating the finitary values $\mathbb{V}$ from the infinitary set($\mathbb{V}$). Thus, our domain does not contain any functions with infinite graphs, but it does contain all the finite approximations.

Theorem 2.2 (Equivalence of relational and denotational semantics). *$\rho \vdash e \Rightarrow v$ iff $v \in \mathcal{E}[\![e]\!]\rho$.*

Proof sketch. The proof is a straightforward induction on e in each direction. □

The following definition lifts the value ordering to environments.

$$\begin{aligned}
X \vdash \rho \sqsubseteq \rho' &\equiv \forall x.\, x \in X \implies \rho(x) \sqsubseteq \rho'(x)\\
\rho \sqsubseteq \rho' &\equiv \forall x.\, \rho(x) \sqsubseteq \rho'(x)
\end{aligned}$$

Proposition 2.3 (Subsumption and Change of Environment).

(1) *If $v \in \mathcal{E}[\![e]\!]\rho$, $v' \sqsubseteq v$, and $\mathrm{fv}(e) \vdash \rho \sqsubseteq \rho'$, then $v' \in \mathcal{E}[\![e]\!]\rho'$.*
(2) *If $v \in \mathcal{E}[\![e]\!]\rho$, $v' \sqsubseteq v$, then $v' \in \mathcal{E}[\![e]\!]\rho$.*

Proof sketch. The proof of part (1) is by induction on e. Part (2) follows directly from (1). □

The proof of Proposition 2.3 is interesting in that it influenced our definition of the semantics. Regarding variables, the semantics for a variable x includes not just the value $\rho(x)$ but also all values below $\rho(x)$ (see Figure 2 or 3). If instead we had defined $\mathcal{E}[\![x]\!]\rho = \rho(x)$, then the case for variables in the proof of Proposition 2.3 would break. The same can be said regarding the result value of function application.

2.3 The Declarative Semantics as a Non-deterministic Interpreter

The denotation function $\mathcal{E}[\![e]\!]\rho$ can also be defined in the style of a non-deterministic interpreter. Recall that a non-deterministic computation returns not just one value, but a set of values. We shall use a monad to manage the non-determinism, making it look, for the most part, like we are writing a normal (deterministic) interpreter (Wadler 1992). The difference compared to a normal interpreter is that monad will give us non-deterministic choice and backtracking. To give meaning to a λ abstraction, we shall non-deterministically choose a finite list of values for its domain and then recursively interpret the body to obtain the corresponding output values. To give meaning to function application, we perform table lookup; if there is a match for the argument value, then we return the output value from the table. If there is not a match, then the interpreter backtracks and tries another table for the λ abstraction.

Non-deterministic choice is provided by the bind operation. It enables us to choose, one at a time, an element a from a sub-computation m, and proceed with another sub-computation f that depends on a, and then collect all the results into a set. The following is the definition of bind together with some short-hand notation to make it convenient to use.

$$\begin{aligned} \mathsf{bind} &: \mathsf{set}(\alpha) \to (\alpha \to \mathsf{set}(\beta)) \to \mathsf{set}(\beta) \\ \mathsf{bind}\; m\; f &= \{b \mid \exists a.\, a \in m \land b \in f(a)\} \\ X \leftarrow m_1; m_2 &\equiv \mathsf{bind}\; m_1\; (\lambda X.\, m_2) \end{aligned}$$

Backtracking is provided by how the zero operation interacts with bind. The zero operation simply says to “fail” or “abort” by returning an empty set.

$$\begin{aligned} \mathsf{zero} &: \mathsf{set}(\alpha) \\ \mathsf{zero} &= \emptyset \end{aligned}$$

In addition to bind and zero, we require one more monadic operation to hide the non-determinism. When a sub-computation finishes with a result, return makes it into a (singleton) set.

$$\begin{aligned} \mathsf{return} &: \alpha \to \mathsf{set}(\alpha) \\ \mathsf{return}\, a &= \{a\} \end{aligned}$$

With these monad operations in hand, we write several auxiliary functions that will be needed for the interpreter. First, we will need a function that non-deterministically chooses a finite set of values, which we write $\mathbb{V}^k$.

$$\begin{aligned} \mathbb{V}^k &: \mathsf{fset}(\mathbb{V}) \\ \mathbb{V}^0 &= \mathsf{return}\; \emptyset \\ \mathbb{V}^{k+1} &= v \leftarrow \mathbb{V}; vs \leftarrow \mathbb{V}^k; \mathsf{return}\; \{v\} \cup vs \end{aligned}$$

The map function applies another, non-deterministic function f, to each element of a finite set, producing a set of finite sets.[3]

$$\begin{aligned} \mathsf{map} &: \mathsf{fset}(\alpha) \to (\alpha \to \mathsf{set}(\beta)) \to \mathsf{set}(\mathsf{fset}(\beta)) \\ \mathsf{map}\; \emptyset\; f &= \mathsf{return}\; \emptyset \\ \mathsf{map}\; (\{a\} \uplus as)\; f &= b \leftarrow f(a); bs \leftarrow \mathsf{map}\; as\; f; \mathsf{return}\; \{b\} \cup bs \end{aligned}$$

The third auxiliary function helps us make sure that the semantics is downward closed with respect to $\sqsubseteq$. The function $\mathsf{down}\, v$ chooses an arbitrary value v' and returns it if $v' \sqsubseteq v$ and otherwise backtracks to pick another value.

$$\begin{aligned} \mathsf{down} &: \mathbb{V} \to \mathsf{set}(\mathbb{V}) \\ \mathsf{down}\; v &= v' \leftarrow \mathbb{V}; \textbf{if}\; v' \sqsubseteq v\; \textbf{then}\; \mathsf{return}\; v'\; \textbf{else}\; \mathsf{zero} \end{aligned}$$

The non-deterministic interpreter for the CBV λ-calculus is defined in Figure 4. Let us focus on the cases for λ abstractions and function application. For $\mathcal{E}[\![\lambda x.\, e]\!]\rho$, we non-deterministically choose a finite domain vs and then map over it to produce the function's table. For each input v, we interpret the body e in an environment extended with v. This produces the output v', which we pair with v to form an entry in the table. For application $\mathcal{E}[\![e_1\; e_2]\!]\rho$, we interpret e_1 and e_2 to the values v_1 and v_2, then check whether v_1 is a function table. If it is, we non-deterministically

[3]We write $\uplus$ for the union of two sets that have no elements in common.

$$\boxed{\mathcal{E}[\![e]\!]\rho}$$

$$
\begin{array}{rl}
\mathcal{E}[\![n]\!]\rho = & \text{return}\, n \\
\mathcal{E}[\![e_1 \oplus e_2]\!]\rho = & v_1 \leftarrow \mathcal{E}[\![e_1]\!]\rho;\; v_2 \leftarrow \mathcal{E}[\![e_2]\!]\rho; \\
& \quad \textbf{case}\,(v_1, v_2)\,\textbf{of}\,(n_1, n_2) \Rightarrow \text{return}\, n_1 \oplus n_2 \\
& \quad \mid \_ \Rightarrow \text{zero} \\
\mathcal{E}[\![x]\!]\rho = & \text{down}\ \rho(x) \\
\mathcal{E}[\![\lambda x.\, e]\!]\rho = & k \leftarrow \mathbb{N}; vs \leftarrow \mathbb{V}^k; \text{map}\ vs\ (\lambda v.\, v' \leftarrow \mathcal{E}[\![e]\!]\rho(x{:=}v); \text{return}\,(v, v')) \\
\mathcal{E}[\![e_1\ e_2]\!]\rho = & v_1 \leftarrow \mathcal{E}[\![e_1]\!]\rho;\; v_2 \leftarrow \mathcal{E}[\![e_2]\!]\rho; \\
& \textbf{case}\, v_1\ \textbf{of}\ \ t \Rightarrow (v, v') \leftarrow t;\ \textbf{if}\, v \sqsubseteq v_2\, \textbf{then}\, \text{down}\ v'\ \textbf{else}\, \text{zero} \\
& \mid \_ \Rightarrow \text{zero} \\
\mathcal{E}[\![\textbf{if}\, e_1\, \textbf{then}\, e_2\, \textbf{else}\, e_3]\!]\rho = & v_1 \leftarrow \mathcal{E}[\![e_1]\!]\rho; \\
& \textbf{case}\, v_1\ \textbf{of}\, n \Rightarrow \textbf{if}\, n \neq 0\, \textbf{then}\, \mathcal{E}[\![e_2]\!]\rho\ \textbf{else}\, \mathcal{E}[\![e_3]\!] \\
& \mid \_ \Rightarrow \text{zero}
\end{array}
$$

Fig. 4. The declarative semantics for CBV λ-calculus as a non-deterministic interpreter.

select an entry (v, v') from the table and see if the argument v_2 matches the parameter v. If so we return down v', otherwise we backtrack and try another entry in the table or possibly backtrack even further and try another table altogether!

The non-deterministic interpreter of Figure 4 is equivalent to the other views of the declarative semantics, though we leave that as an exercise to the reader.

2.4 The Declarative Semantics as a Type System

The declarative semantics can be viewed from a rather different angle: as a type system with intersection and singleton integer types. It may sound impossible to have a semantics that is equivalent to a type system, but this type system is unusual in its precision. It is so precise that it is undecidable, which in some sense makes it a bad type system. It also may seem strange to be giving a type system for the untyped λ-calculus. But this type system comes from the Curry school of super-imposing type systems on untyped languages (Curry and Feys 1958). Nevertheless, for readers who are most comfortable with the notation of type systems, the semantics as type-system presentation can be helpful. Also, this connection helps in the proof of correctness (Section 3.2) where we adapt results from the literature on intersection types.

The grammar of types is the following and consists of two non-terminals, one for types of functions and one for types in general.

$$
\begin{array}{rcll}
F, G, H & ::= & A \to B \mid F \wedge G \mid \top & \text{types of functions} \\
A, B, C \in \mathbb{T} & ::= & n \mid F & \text{types}
\end{array}
$$

The type n is the singleton integer type inhabited by just n. The intersection type $G \wedge H$ is for functions that have type F and G. All functions have type $\top$[4]. The use of two non-terminals enables the restriction of the intersection types to not include singleton integer types, which would either be trivial (e.g. $1 \wedge 1$) or garbage (e.g. $0 \wedge 1$). (This stratification is a minor point of difference with the literature on intersection types.)

[4]Our $\top$ type is sometimes written ν in the literature (Egidi et al. 1992).

$$\boxed{A <: B} \quad \boxed{F <: G}$$

$$\frac{}{A <: A} \qquad \frac{A <: B \quad B <: C}{A <: C} \qquad \frac{A \approx A' \quad B \approx B'}{A \to B <: A' \to B'}$$

$$\frac{H <: F \quad H <: G}{H <: F \wedge G} \qquad \frac{}{F \wedge G <: F} \qquad \frac{}{F \wedge G <: G} \qquad \frac{}{A \to B <: \top}$$

$$A \approx B \equiv \; A <: B \text{ and } B <: A$$

Fig. 5. A subtyping relation on intersection types.

We define subtyping <: and type equivalence ≈ in Figure 5 but with fewer rules than is typical (Barendregt et al. 2013; Pierce 2002). We omit the following rules, i.e., the subtyping rule for functions and for distributing intersections through functions.

$$\frac{A' <: A \quad B <: B'}{A \to B <: A' \to B'} \qquad \frac{}{(C \to A) \wedge (C \to B) <: C \to (A \wedge B)}$$

The types we have defined are isomorphic to the values from the beginning of Section 2. The singleton types are isomorphic to integers. The function, intersection, and $\top$ types taken together are isomorphic to function tables. Each entry in a table corresponds to a function type. The following two functions, typof and valof, witness the isomorphism. Strictly speaking, the isomorphism is between $\mathbb{V}$ and $\mathbb{T}/\approx$, so values are a canonical form for types.

$$\begin{aligned} \mathsf{typof} &: \mathbb{V} \to \mathbb{T} \\ \mathsf{typof}(n) &= n \\ \mathsf{typof}(t) &= \bigwedge_{(v,v')\in t} \mathsf{typof}(v) {\to} \mathsf{typof}(v') \end{aligned} \qquad \begin{aligned} \mathsf{valof} &: \mathbb{T} \to \mathbb{V} \\ \mathsf{valof}(n) &= \{n\} \\ \mathsf{valof}(F) &= \mathsf{tabof}(F) \end{aligned}$$

where

$$\begin{aligned} \mathsf{tabof}(A \to B) &= \{(\mathsf{valof}(A), \mathsf{valof}(B))\} \\ \mathsf{tabof}(A \wedge B) &= \mathsf{valof}(A) \cup \mathsf{valof}(B) \\ \mathsf{tabof}(\top) &= \emptyset \end{aligned}$$

It is relatively easy to see that this subtyping relation is the inverse of the $\sqsubseteq$ ordering on values.

Proposition 2.4 (Subtyping is inverse of $\sqsubseteq$ and is related to $\in$).

(1) *If* $A <: B$, *then* $\mathsf{valof}(B) \sqsubseteq \mathsf{valof}(A)$.
(2) *If* $v \sqsubseteq v'$, *then* $\mathsf{typof}(v') <: \mathsf{typof}(v)$.
(3) *If* $(v, v') \in t$, *then* $\mathsf{typof}(t) <: \mathsf{typof}(v) \to \mathsf{typof}(v')$.

Proposition 2.5 (Types and Values are Isomorphic).
$\mathsf{valof}(\mathsf{typof}(v)) = v$ *and* $\mathsf{typof}(\mathsf{valof}(A)) \approx A$

Figure 6 defines the declarative semantics as a type system. As usual, the *type environment* Γ is a partial mapping from variables to types. The type system includes a subsumption rule and the standard rules for a simply-typed λ-calculus, except that an integer is assigned the singleton type for itself. The type system does not have the standard rule for intersection introduction:

$$\frac{\Gamma \vdash e : A \quad \Gamma \vdash e : B}{\Gamma \vdash e : A \wedge B}$$

$$\boxed{\Gamma \vdash e : A}$$

$$\frac{}{\Gamma \vdash n : n} \qquad \frac{\Gamma \vdash e_1 : n_1 \quad \Gamma \vdash e_2 : n_2}{\Gamma \vdash e_1 \oplus e_2 : n_1 \oplus n_2} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma, x{:}A \vdash e : B}{\Gamma \vdash \lambda x.\, e : A \to B} \qquad \frac{}{\Gamma \vdash \lambda x.\, e : \top} \qquad \frac{\Gamma \vdash \lambda x.\, e : A \quad \Gamma \vdash \lambda x.\, e : B}{\Gamma \vdash \lambda x.\, e : A \wedge B}$$

$$\frac{\Gamma \vdash e_1 : A \to B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B} \qquad \frac{\Gamma \vdash e : A \quad A <: B}{\Gamma \vdash e : B}$$

$$\frac{\Gamma \vdash e_1 : n \quad n \neq 0 \quad \Gamma \vdash e_2 : B}{\Gamma \vdash \textbf{if}\, e_1\, \textbf{then}\, e_2\, \textbf{else}\, e_3 : B} \qquad \frac{\Gamma \vdash e_1 : n \quad n = 0 \quad \Gamma \vdash e_3 : B}{\Gamma \vdash \textbf{if}\, e_1\, \textbf{then}\, e_2\, \textbf{else}\, e_3 : B}$$

Fig. 6. The declarative semantics for CBV λ-calculus as a type system.

but instead requires e to be a λ abstraction. The same holds true of the rule for $\top$ introduction.

Theorem 2.6 (Equivalence of type system and denotational semantics).

(1) $v \in \mathcal{E}[\![e]\!]\rho$ *implies* $\mathrm{typof}(\rho) \vdash e : \mathrm{typof}(v)$ *and*
(2) $\Gamma \vdash e : A$ *implies* $\mathrm{valof}(A) \in \mathcal{E}[\![e]\!]\mathrm{valof}(\Gamma)$.

where $\mathrm{typof}(\rho)(x) = \mathrm{typof}(\rho(x))$ *and* $\mathrm{valof}(\Gamma)(x) = \mathrm{valof}(\Gamma(x))$.

3 CORRECTNESS OF THE DECLARATIVE SEMANTICS

We prove that the declarative semantics for CBV λ-calculus is equivalent to the standard operational semantics on programs. For this section and the next we use the denotational version of the declarative semantics. Section 3.1 proves one direction of the equivalence and Section 3.2 proves the other. Then to justify using the declarative semantics for compiler optimizations, we prove in Section 3.3 that it is sound with respect to contextual equivalence.

3.1 Sound wrt. Operational Semantics

In this section we prove Theorem 1, in particular, that the declarative semantics is sound with respect to the relational semantics of Kahn (1987), which we discussed in Section 2.1. For each lemma and theorem we give a proof sketch that includes which other lemmas or theorems were needed. For the full details we refer the reader to the Isabelle mechanization.

We relate the values of the declarative semantics to sets of values of Kahn (1987) with the following logical relation. Logical relations are usually type-indexed, not value-indexed, but our values are isomorphic to types.

$$\begin{aligned}
&\mathcal{G} : \mathbb{V} \to \mathrm{set}(\mathbb{W})\\
&\mathcal{G}(n) = \{n\}\\
&\mathcal{G}(t) = \{\langle \lambda x.\, e, \varrho\rangle \mid \forall (v_1, v_2){\in}t, w_1.\, w_1{\in}\mathcal{G}(v_1) \implies \exists w_2.\, \varrho(x{:=}w_1) \vdash_s e \Rightarrow w_2 \wedge w_2{\in}\mathcal{G}(v_2)\}
\end{aligned}$$

Lemma 3.1 ($\mathcal{G}$ is downward closed). *If* $w \in \mathcal{G}(v)$ *and* $v' \sqsubseteq v$, *then* $w \in \mathcal{G}(v')$.

Proof sketch. The proof is a straightforward induction on v. □

We relate the environments of the two semantics with the inductively defined predicate $\mathcal{G}(\rho, \varrho)$.

$$\frac{}{\mathcal{G}(\emptyset, \emptyset)} \qquad \frac{w \in \mathcal{G}(v) \quad \mathcal{G}(\rho, \varrho)}{\mathcal{G}(\rho(x{:=}v), \varrho(x{:=}w))}$$

Lemma 3.2. *If $\mathcal{G}(\rho, \varrho)$, then $\varrho(x) \in \mathcal{G}(\rho(x))$*

Proof sketch. The proof is a straightforward induction on the derivation of $\mathcal{G}(\rho, \varrho)$. □

Lemma 3.3. *If $v \in \mathcal{E}[\![e]\!]\rho$ and $\mathcal{G}(\rho, \varrho)$, then $\varrho \vdash_s e \Rightarrow w$ and $w \in \mathcal{G}(v)$ for some w.*

Proof sketch. The proof is by induction on e. The cases for integers and arithmetic operations are straightforward. The case for variables uses Lemmas 3.1 and 3.2. The cases for λ abstraction and application use Lemma 3.1 but are otherwise straightforward. □

Theorem 1 (Sound wrt. op. sem.). *(aka. Adequacy) If $\mathcal{E}[\![e]\!]\emptyset = \mathcal{E}[\![n]\!]\emptyset$, then $e \Downarrow n$.*

Proof sketch. From the premise we have $n \in \mathcal{E}[\![e]\!]\emptyset$. Also we immediately have $\mathcal{G}(\emptyset, \emptyset)$. So by Lemma 3.3 we have $\emptyset \vdash_s e \Rightarrow w$ and $w \in \mathcal{G}(n)$ for some w. So $w = n$ and therefore $e \Downarrow n$. □

This proof of soundness for the declarative semantics can also be viewed as a proof of completeness for the operational semantics. The operational semantics, being operational, can be viewed as a kind of implementation, and in this light, the above proof is an example of how convenient it can be to prove (one direction of) implementation correctness using the declarative semantics as the specification.

3.2 Complete wrt. Operational Semantics

In this section we prove Theorem 2, in particular, that the declarative semantics is complete with respect to the small-step semantics for the CBV λ-calculus (Pierce 2002). The proof strategy is adapted from work by Alessi et al. (2003) on intersection types. We need to show that if $e \longrightarrow^* n$, then $\mathcal{E}[\![e]\!]\emptyset = \mathcal{E}[\![n]\!]\emptyset$. It is obviously true that meaning of the last expression in the reduction sequence is equal to $\mathcal{E}[\![n]\!]\emptyset$, so if we could just walk this backwards, one step at a time, we would have our result. That is, we need to show that if $e \longrightarrow e'$, then $\mathcal{E}[\![e]\!]\rho = \mathcal{E}[\![e']\!]\rho$. We can decompose the equality into $\mathcal{E}[\![e]\!]\rho \subseteq \mathcal{E}[\![e']\!]\rho$ and $\mathcal{E}[\![e']\!]\rho \subseteq \mathcal{E}[\![e]\!]\rho$. The forward direction is equivalent to proving "preservation" for our intersection type system, which is straightforward. Let us focus on the backward direction in the case of β reduction. Consider the following example reduction:

$$(\lambda x.\, (x\ 1) + (x\ 2))\ (\lambda y.\ \ldots) \longrightarrow ((\lambda y.\ \ldots)\ 1) + ((\lambda y.\ \ldots)\ 2)$$

where $e = (\lambda x.\ \ldots)\ (\lambda y.\ \ldots)$ and $e' = ((\lambda y.\ \ldots)\ 1) + ((\lambda y.\ \ldots)\ 2)$. For some arbitrary v, we can assume that $v \in \mathcal{E}[\![e']\!]\emptyset$ and need to show $v \in \mathcal{E}[\![e]\!]\emptyset$. From $v \in \mathcal{E}[\![e']\!]\emptyset$ we know there must have been some tables t_1 and t_2 such that $t_1 \in \mathcal{E}[\![\lambda y.\ \ldots]\!]\emptyset$ and $t_2 \in \mathcal{E}[\![\lambda y.\ \ldots]\!]\emptyset$, but we only know for sure that 1 is in the domain of t_1 and 2 is in the domain of t_2. Perhaps $t_1 = \{(1, 7)\}$ and $t_2 = \{(2, 0)\}$. However, to obtain $v \in \mathcal{E}[\![e]\!]\emptyset$ we need a single table t_3 that can be bound to variable x, and has both 1 and 2 in its domain so that the applications $(x\ 1)$ and $(x\ 2)$ make sense. Fortunately, we can simply combine the two tables:

$$t_3 = t_1 \cup t_2 = \{(1, 7), (2, 0)\} \tag{1}$$

In general, we will need a lemma about reverse substitution. Recall the rule for β reduction

$$(\lambda x.\, e_1)\ e_2 \longrightarrow e_1[e_2/x]$$

Then we need a lemma that says something like, for any v_1,

$$v_1 \in \mathcal{E}[\![e_1[e_2/x]]\!]\emptyset \quad \text{implies} \quad v_1 \in \mathcal{E}[\![e_1]\!]\emptyset(x{:=}v_2)$$

for some $v_2 \in \mathcal{E}[\![e_2]\!]\emptyset$. Generalizing this so that it can be proved by induction takes some work but gives us Lemma 3.6 below.

We proceed with the formal development of the completeness proof. We lift the union operation to values with the following join operator.

$$n \sqcup n = n \qquad\qquad t \sqcup t' = t \cup t' \qquad\qquad \boxed{v \sqcup v}$$

The join is not always defined, for example, there is no join of the integers 0 and 1.

Proposition 3.4 (Join is the least upper bound of $\sqsubseteq$).

(1) *$v_1 \sqsubseteq v_1 \sqcup v_2$, $v_2 \sqsubseteq v_1 \sqcup v_2$, and*
(2) *If $v_1 \sqsubseteq v_3$ and $v_2 \sqsubseteq v_3$, then $v_1 \sqcup v_2 \sqsubseteq v_3$.*

We write $\overline{e}$ for *syntactic values* that are closed:

$$\overline{e} ::= n \mid \lambda x.\, e \qquad \text{where } \mathrm{fv}(\lambda x.\, e) = \emptyset$$

The following lemma says that, if we have two different values v_1, v_2 in the meaning of a syntactic value $\overline{e}$, then their join is also in the meaning of $\overline{e}$. This lemma generalizes the way in which we combined t_1 and t_2 into t_3 in equation (1).

Lemma 3.5 (Joining Values). *If $v_1 \in \mathcal{E}[\![\overline{e}]\!]\rho$ and $v_2 \in \mathcal{E}[\![\overline{e}]\!]\rho$, then $v_1 \sqcup v_2 \in \mathcal{E}[\![\overline{e}]\!]\rho$.*

Proof sketch. The proof is a straightforward induction on $\overline{e}$. □

We define equivalence of environments, written $\rho \approx \rho'$, and note that $\rho \approx \rho'$ implies $\rho \sqsubseteq \rho'$.

$$\rho \approx \rho' \equiv \forall x.\, \rho(x) = \rho'(x)$$

Lemma 3.6 (Reverse substitution preserves meaning). *If $v \in \mathcal{E}[\![e[\overline{e}/y]]\!]\rho$, then $v \in \mathcal{E}[\![e]\!]\rho'$, $v' \in \mathcal{E}[\![\overline{e}]\!]\emptyset$, and $\rho' \approx \rho(y{:=}v')$ for some ρ', v'.*

Proof sketch. The proof is by induction on $e' = e[\overline{e}/y]$. But before considering the cases for e', we first consider whether or not $e = y$. The proof uses Propositions 2.3 and 3.12 and Lemma 3.5. □

Lemma 3.7 (Reverse reduction preserves meaning).

(1) *If $e \longrightarrow e'$, then $\mathcal{E}[\![e']\!]\rho \subseteq \mathcal{E}[\![e]\!]\rho$.*
(2) *If $e \longrightarrow^* e'$, then $\mathcal{E}[\![e']\!]\rho \subseteq \mathcal{E}[\![e]\!]\rho$.*

Proof sketch.

(1) The proof is by induction on the derivation of $e \longrightarrow e'$. All of the cases are straightforward except for β reduction. In that case we have $(\lambda x.\, e_1)\ \overline{e} \longrightarrow e_1[\overline{e}/x]$. Fix an arbitrary v and assume $v \in \mathcal{E}[\![e_1[\overline{e}/x]]\!]\rho$. We need to show that $v \in \mathcal{E}[\![(\lambda x.\, e_1)\ \overline{e}]\!]\rho$. By Lemma 3.6 and the assumption there exist v' and ρ' such that $v \in \mathcal{E}[\![e_1]\!]\rho'$, $v' \in \mathcal{E}[\![\overline{e}]\!]\emptyset$, and $\rho' \approx \rho(x{:=}v')$. Then we have $v \in \mathcal{E}[\![e_1]\!]\rho(x{:=}v')$ by Proposition 2.3, noting that $\rho' \sqsubseteq \rho(x{:=}v')$. Therefore we have $\{(v', v)\} \in \mathcal{E}[\![\lambda x.\, e_1]\!]\rho$. Also, we have $v' \in \mathcal{E}[\![\overline{e}]\!]\rho$ by another use of Proposition 2.3, noting that $\emptyset \sqsubseteq \rho$. With these two facts, we conclude that $v \in \mathcal{E}[\![(\lambda x.\, e_1)\ \overline{e}]\!]\rho$.
(2) The proof is by induction on the derivation of $e \longrightarrow^* e'$. The base case is trivial and the induction step follows immediately from part (1).

□

Next we prove the forward direction, that reduction preserves meaning. The proof follows the usual pattern for "preservation" of a type system. However, we shall continue to use the denotational notation, writing $v \in \mathcal{E}[\![e]\!]\rho$, rather than $\Gamma \vdash e : A$.

Lemma 3.8 (Substitution preserves meaning). *If* $v \in \mathcal{E}[\![e]\!]\rho'$, $v' \in \mathcal{E}[\![\bar{e}]\!]\emptyset$, *and* $\rho' \approx \rho(x{:=}v')$, *then* $v \in \mathcal{E}[\![e[\bar{e}/x]]\!]$

Proof sketch. The proof is by induction on e. The case for variables uses Proposition 2.3. The case for λ uses Proposition 2.3. □

Lemma 3.9 (Reduction preserves meaning).

(1) *If* $e \longrightarrow e'$, *then* $\mathcal{E}[\![e]\!]\rho \subseteq \mathcal{E}[\![e']\!]\rho$.
(2) *If* $e \longrightarrow^* e'$, *then* $\mathcal{E}[\![e]\!]\rho \subseteq \mathcal{E}[\![e']\!]\rho$.

Proof sketch.

(1) The proof is by induction on $e \longrightarrow e'$. Most of the cases are straightforward. The case for β uses Proposition 2.3 and Lemma 3.8.
(2) The proof is by induction on the derivation of $e \longrightarrow^* e'$, using part (1) in the induction step.

□

Corollary 3.10 (Meaning is invariant under reduction).
If $e \longrightarrow^* e'$, *then* $\mathcal{E}[\![e]\!] = \mathcal{E}[\![e']\!]$.

Proof sketch. The two directions of the equality are proved by Lemma 3.7 and 3.9. □

Theorem 2 (Complete wrt. op. sem.). *If* $e \Downarrow n$, *then* $\mathcal{E}[\![e]\!]\emptyset = \mathcal{E}[\![n]\!]\emptyset$.

Proof sketch. From the premise we have $e \longrightarrow^* n$, from which we conclude by use of Lemma 3.10. □

The completeness theorem is rather important for the declarative semantics. It says that $\mathcal{E}$ gives the right meaning to all the terminating programs in the CBV λ-calculus.

We also know that $\mathcal{E}$ gives the right meaning to diverging programs. That is, $\mathcal{E}$ maps diverging programs to $\emptyset$. We write $e \Uparrow$ when e diverges.

Proposition 3.11 (Diverging programs have empty meaning). *If* $e \Uparrow$ *then* $\mathcal{E}[\![e]\!]\emptyset = \emptyset$.

Proof sketch. Towards a contradiction, suppose $\mathcal{E}[\![e]\!]\emptyset \neq \emptyset$. So $v \in \mathcal{E}[\![e]\!]\emptyset$ for some v We have $\mathcal{G}(\emptyset, \emptyset)$, so $\emptyset \vdash e \Rightarrow v$ by Lemma 3.3. Thus, we also have $e \longrightarrow^* \bar{e}$. Then from $e \Uparrow$, we have $\bar{e} \longrightarrow e'$ for some e', but that is impossible because $\bar{e}$ is a value and so cannot further reduce. □

In contrast, syntactic values have non-empty meaning.

Proposition 3.12 (Syntactic values have non-empty meaning). $\mathcal{E}[\![\bar{e}]\!]\rho \neq \emptyset$.

Proof sketch. The proof is by case analysis on $\bar{e}$. □

Our proof of completeness related the declarative semantics to the small-step operational semantics. We wonder whether a more elegant proof is possible using big-step operational semantics, similar to our proof of soundness. Hopefully the insights we have gained in the above proof, especially Lemma 3.8, will provide some guidance.

3.3 Sound wrt. Contextual Equivalence

Recall that we want to use the declarative semantics to justify compiler optimizations, which replace sub-expressions within a program with other sub-expressions (that are hopefully more efficient). Two sub-expressions are *contextual equivalent*, defined below, when replacing one with the other does not change the behavior of the program (Morris 1968). We define contexts C with the following grammar.

$$C ::= C \oplus e \mid e \oplus C \mid \lambda x.\, C \mid C\ e \mid e\ C \qquad \text{contexts}$$
$$e \simeq e' \equiv \forall C.\, FV(C[e]) = FV(C[e']) = \emptyset \implies C[e] \Downarrow \text{ iff } C[e'] \Downarrow \qquad \text{contextual equivalence}$$

The correctness property that we are after is that denotational equality should imply contextual equivalence. A common way to prove this is to show that the denotational semantics is a congruence and then use soundness and completeness of the semantics for programs (Gunter 1992).

Lemma 3.13 (E is a congruence). *For any context C, if $\mathcal{E}[\![e]\!] = \mathcal{E}[\![e']\!]$, then $\mathcal{E}[\![C[e]]\!] = \mathcal{E}[\![C[e']]\!]$.*

Proof sketch. The proof is a straightforward induction on C. □

Theorem 3 (Sound wrt. Contextual Equivalence).
If $\mathcal{E}[\![e]\!] = \mathcal{E}[\![e']\!]$, then $C[e] \Downarrow$ iff $C[e'] \Downarrow$ for any closing context C.

Proof sketch. We discuss one direction of the iff, that $C[e] \Downarrow$ implies $C[e'] \Downarrow$. The other direction is similar. From the premise, congruence gives us $\mathcal{E}[\![C[e]]\!] = \mathcal{E}[\![C[e']]\!]$. From $C[e] \Downarrow$ we have $C[e] \longrightarrow^* \bar{e}$ for some $\bar{e}$. Therefore we have $\mathcal{E}[\![C[e]]\!]\emptyset = \mathcal{E}[\![\bar{e}]\!]\emptyset$ by completeness (Theorem 2). Then we also have $\mathcal{E}[\![C[e']]\!]\emptyset = \mathcal{E}[\![\bar{e}]\!]$. So by Proposition 3.12, we have $v \in \mathcal{E}[\![\bar{e}]\!]$ for some v, and therefore $v \in \mathcal{E}[\![C[e']]\!]$. We conclude that $\emptyset \vdash C[e'] \Rightarrow w$ for some w by soundness (Lemma 3.3). □

4 CORRECTNESS OF AN OPTIMIZER

We turn to address the question of whether the declarative semantics is useful. Our first case study is proving the correctness of a compiler optimization pass: constant folding and function inlining. The setting is still the untyped λ-calculus extended with integers and arithmetic. Figure 7 defines the optimizer as a function $\mathcal{O}$ that maps an expression and a counter to an expression. One of the challenging problems in creating a good inliner is determining when to stop. Here we simply use a counter that limits inlining to a fixed depth k. A real optimizing compiler would use a smarter heuristic but it would employ the same program transformations.

The third equation in the definition of $\mathcal{O}$ performs constant folding. For an arithmetic operation $e_1 \oplus e_2$, it recursively optimizes e_1 and e_2. If the results are integers, then it performs the arithmetic. Otherwise it outputs an arithmetic expression to perform the arithmetic at runtime. The fourth equation optimizes the body of a λ abstraction. The fifth equation, for function application, is the most interesting. If e_1 optimizes to a λ abstraction and e_2 optimizes to a syntactic value v_2, then we perform inlining by substituting v_2 for parameter x in the body of the function. We then optimize the result of the substitution, making this a rather aggressive polyvariant optimizer (Banerjee 1997; Jagannathan and Wright 1996; Turbak et al. 1997; Waddell and Dybvig 1997). The counter is decremented on this recursive call to ensure termination.

We turn to proving the optimizer correct with respect to the declarative semantics. The proof is pleasantly straightforward!

Lemma 4.1 (Optimizer Preserves Denotations). $\mathcal{E}(\mathcal{O}[\![e]\!]k) = \mathcal{E}[\![e]\!]$

Proof sketch. The proof is by induction on the termination metric for $\mathcal{O}$, which is the lexicographic ordering of k then the size of e. All the cases are straightforward to prove because reduction preserves meaning (Lemma 3.10) and because meaning is a congruence (Lemma 3.13). □

$$\begin{aligned}
O[\![x]\!]k &= x \\
O[\![n]\!]k &= n \\
O[\![e_1 \oplus e_2]\!]k &= \begin{cases} n_1 \oplus n_2 & \text{if } O[\![e_1]\!]k = n_1 \text{ and } O[\![e_2]\!]k = n_2 \\ [\![O[\![e_1]\!]k \oplus O[\![e_2]\!]k]\!] & \text{otherwise} \end{cases} \\
O[\![\lambda x.\, e]\!]k &= \lambda x.\, O[\![e]\!]k \\
O[\![e_1\ e_2]\!]k &= \begin{cases} O[\![e[v_2/x]]\!](k{-}1) & \text{if } k \geq 1 \text{ and } O[\![e_1]\!]k = \lambda x.\, e \\ & \text{and } O[\![e_2]\!]k = v_2 \\ [\![O[\![e_1]\!]k\ O[\![e_2]\!]k]\!] & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 7. A compiler optimization pass that folds and propagates constants and inlines function calls.

$$\begin{array}{rcll}
i, j & \in & \mathbb{N} & \text{DeBruijn indices} \\
A, B, C & ::= & \texttt{int} \mid A \to B \mid \forall\, A \mid i & \text{types} \\
e & ::= & n \mid e \oplus e \mid x \mid \lambda x{:}A.\, e \mid e\ e \mid \Lambda e \mid e[A] \mid \texttt{fix}\, x{:}A.\, e & \text{expressions}
\end{array}$$

$$\Gamma \vdash n : \texttt{int} \qquad \frac{}{\Gamma \vdash x : \text{lookup}(\Gamma, x)}$$

$$\frac{\text{extend}(\Gamma, x, A) \vdash e : B}{\Gamma \vdash \lambda x{:}A.\, e : A \to B} \qquad \frac{\text{extend}(\Gamma, x, A \to B) \vdash e : A \to B}{\Gamma \vdash \texttt{fix}\, x{:}A{\to}B.\, e : A \to B} \qquad \frac{\Gamma \vdash e : A \to B \quad \Gamma \vdash e' : A}{\Gamma \vdash e\ e' : B}$$

$$\frac{\text{tyExtend}(\Gamma) \vdash e : A}{\Gamma \vdash \Lambda e : \forall A} \qquad \frac{\Gamma \vdash e : \forall A}{\Gamma \vdash e[B] : [0 \mapsto B]A}$$

Fig. 8. Syntax and type system of System F extended with general recursion (via `fix`).

The mechanized proof of Lemma 4.1 is only 16 lines!

Theorem 4.2 (Correctness of the Optimizer). $e \simeq O[\![e]\!]k$

Proof sketch. The proof follows directly from the above Lemma 4.1 and soundness with respect to contextual equivalence (Theorem 3). □

5 SEMANTICS AND TYPE SOUNDNESS FOR SYSTEM F

This section serves two purposes: it demonstrates that the declarative semantics is straightforward to extend to languages with first-class polymorphism, and it demonstrates how to use the declarative semantics to prove type soundness. Our setting is the polymorphic λ-calculus (System F) (Girard 1972; Reynolds 1974) extended with general recursion (`fix`).

5.1 Static Semantics

The syntax and type system of this language is defined in Figure 8. We choose to use DeBruijn indices for type variables, with $\forall$ and Λ acting as implicit binding forms for type variables. The shift operation $\uparrow^k_c (A)$ increases the DeBruijn indices greater or equal to c in type A by k and the substitution operation $[i \mapsto B]A$ replaces DeBruijn index i within type A with type B (Pierce 2002).

Type environments and their operations deserve some explanation in the way they handle type variables. A *type environment* Γ is a pair consisting of 1) a mapping for term variables and 2) a natural number representing the number of enclosing type-variable binders. The mapping is from variables names to a pair of a) the variable's type and b) the number of enclosing type-variable binders at the variable's point of definition. We define the operations extend, tyExtend, and lookup in the following way.

$$\begin{aligned}\mathsf{extend}(\Gamma, x, A) &\equiv (\mathsf{fst}(\Gamma)(x{:=}(A, \mathsf{snd}(\Gamma)),\ \mathsf{snd}(\Gamma))\\ \mathsf{tyExtend}(\Gamma) &\equiv (\mathsf{fst}(\Gamma),\ \mathsf{snd}(\Gamma) + 1)\\ \mathsf{lookup}(\Gamma, x) &\equiv \uparrow_0^{k-j} (A) \qquad\qquad \text{where } (A, j) = \mathsf{fst}(\Gamma)(x) \text{ and } k = \mathsf{snd}(\Gamma)\end{aligned}$$

Thus, the lookup operation properly transports the type of a variable from its point of definition to its occurrence by shifting its type variables the appropriate amount. We say that a type environment Γ is *well-formed* if $\mathsf{snd}(\mathsf{fst}(\Gamma)(x)) \le \mathsf{snd}(\Gamma)$ for all $x \in \mathsf{dom}(\mathsf{fst}(\Gamma))$.

5.2 Dynamic Semantics

The domain of values differs from that of the λ-calculus in two respects. First, we add a wrong value to represent a runtime type error so that we can prove that "well-typed programs cannot go wrong" in the tradition of Milner (1978). Second, we add a value to represent type abstraction. From a runtime point of view, a type abstraction Λe simply produces a thunk. That is, it delays the execution of expression e until the point of type application (aka. instantiation). A thunk contains an optional value, that is, if the expression e does not terminate, then the thunk contains none, otherwise the thunk contains a value produced by e.

$$\begin{array}{rcll} o & ::= & \mathsf{none} \mid \mathsf{some}(v) & \text{optional values}\\ v & ::= & n \mid t \mid \mathsf{thunk}(o) \mid \mathsf{wrong} & \text{values}\end{array}$$

Building on the non-determinism monad that we introduced in Section 2.3, we add support for the short-circuiting due to errors with the following alternative form of bind.

$$X := E_1; E_2 \equiv X \leftarrow E_1; \textbf{if } X = \mathsf{wrong}\ \textbf{then}\ \mathsf{return\ wrong}\ \textbf{else}\ E_2$$

We define an auxiliary function apply to give the semantics of function application. The main difference regarding function application with respect to Figure 4 is returning wrong when e_1 produces a value that is not a function table. We also replace uses of $\leftarrow$ with $:=$ to short-circuit the computation in case wrong is produced by a sub-computation.

$$\begin{aligned}\mathsf{apply}(V_1, V_2) = \ & v_1 := V_1;\ v_2 := V_2;\\ & \textbf{case}\ v_1\ \textbf{of}\ \ t \Rightarrow (v, v') \leftarrow t;\ \ \textbf{if}\ v \sqsubseteq v_2\ \textbf{then}\ \mathsf{down}\ v'\ \textbf{else}\ \mathsf{zero}\\ & \mid \_ \Rightarrow \mathsf{return\ wrong}\end{aligned}$$

The declarative semantics for System F extended with `fix` is defined in Figure 9. Regarding integers and arithmetic, the only difference is that arithmetic operations return wrong when an input in not an integer. Regarding λ abstraction, the semantics is the same as for the untyped λ-calculus (Figure 4).

To give meaning to (`fix` $x{:}A.\ e$) we form its ascending Kleene chain but never take its supremum. In other words, we define a auxiliary function iterate that starts with no tables ($\emptyset$) and then iteratively feeds the function to itself some finite number of times, produces ever-larger sets of tables that better approximate the function. The parameter L is the meaning function $\mathcal{E}$ partially applied to the expression e. We fully apply L inside of iterate, binding x to the previous approximations.

The declarative semantics of polymorphism is straightforward. The meaning of a type abstraction Λe is $\mathsf{thunk}(\mathsf{some}(v))$ if e evaluates to v. The meaning of Λe is $\mathsf{thunk}(\mathsf{none})$ if e diverges. The meaning of a type application $e[A]$ is to force the thunk, that is, it is any value below v if

$$
\begin{aligned}
\mathcal{E}[\![n]\!]\rho &= \mathsf{return}\, n \\
\mathcal{E}[\![e_1 \oplus e_2]\!]\rho &= \; v_1 := \mathcal{E}[\![e_1]\!]\rho;\; v_2 := \mathcal{E}[\![e_2]\!]\rho; \\
&\quad\; \mathbf{case}\,(v_1, v_2)\,\mathbf{of}\,(n_1, n_2) \Rightarrow \mathsf{return}\, n_1 \oplus n_2 \mid \_ \Rightarrow \mathsf{return\, wrong} \\
\mathcal{E}[\![x]\!]\rho &= \mathsf{down}\, \rho(x) \\
\mathcal{E}[\![\lambda x{:}A.\, e]\!]\rho &= k \leftarrow \mathbb{N}; vs \leftarrow \mathbb{V}^k; \mathsf{map}\; vs\; (\lambda v.\, v' := \mathcal{E}[\![e]\!]\rho(x{:=}v); \mathsf{return}\,(v, v')) \\
\mathcal{E}[\![e_1\; e_2]\!]\rho &= \mathsf{apply}(\mathcal{E}[\![e_1]\!]\rho, \mathcal{E}[\![e_2]\!]\rho) \\
\mathcal{E}[\![\mathtt{fix}\, x{:}A.\, e]\!]\rho &= k \leftarrow \mathbb{N}; \mathsf{iterate}(k, x, \mathcal{E}[\![e]\!], \rho) \\
&\qquad\qquad \mathbf{where}\;\; \mathsf{iterate}(0, x, L, \rho) = \mathsf{zero} \\
&\qquad\qquad\qquad\quad\;\; \mathsf{iterate}(k+1, x, L, \rho) = v := \mathsf{iterate}(k, x, L, \rho); L\, \rho(x{:=}v) \\
\mathcal{E}[\![\Lambda e]\!]\rho &= \; \mathbf{if}\, \mathcal{E}[\![e]\!]\rho = \emptyset\, \mathbf{then}\, \mathsf{return\, thunk(none)} \\
&\quad\; \mathbf{else}\, v := \mathcal{E}[\![e]\!]\rho; \mathsf{return\, thunk(some}(v)) \\
\mathcal{E}[\![e[A]]\!]\rho &= x := \mathcal{E}[\![e]\!]\rho;\;\; \mathbf{case}\, x\, \mathbf{of}\, \mathsf{thunk(none)} \Rightarrow \mathsf{zero} \\
&\qquad\qquad\qquad\quad\; \mid \mathsf{thunk(some}(v)) \Rightarrow \mathsf{down}\, v \\
&\qquad\qquad\qquad\quad\; \mid \_ \Rightarrow \mathsf{return\, wrong}
\end{aligned}
$$

Fig. 9. The declarative semantics for System F.

e evaluates to thunk(some(v)). On the other hand, if e evaluates to thunk(none), then the type application diverges. Finally, if e evaluates to something other than a thunk, the result is wrong.

One strength of the declarative semantics is that it enables the use of monads to implicitly handle error propagation, which is important for scaling up to large language specifications (Amin and Rompf 2017; Charguéraud 2013; Owens et al. 2016).

5.3 Semantics of Types

We define types as sets of values with the meaning function $\mathcal{T}$ defined in Figure 10. The meaning of `int` is the integers. To handle type variables, $\mathcal{T}$ has a second parameter η that maps each DeBruijn index to a set of values, i.e., the meaning of a type variable. So the meaning of index i is basically η_i. The cleanup function serves to make sure that $\mathcal{T}$ is downward closed and that it does not include wrong. (Putting the use of cleanup in $\mathcal{T}[\![i]\!]$ instead of $\mathcal{T}[\![\forall A]\!]$ makes for a slightly simpler definition of well-typed environments, which is defined below.) We write $|\eta|$ for the length of a sequence η. The meaning of a function type $A \to B$ is all the finite tables t in which those entries with input value in $\mathcal{T}[\![A]\!]$ have output value in $\mathcal{T}[\![B]\!]$.

Last but not least, the meaning of a universal type $\forall A$ includes thunk(none) but also thunk(some(v)) whenever v is in the meaning of A but with η extended with an arbitrary set of values.

We write $\vdash \rho, \eta : \Gamma$ to say that environments ρ and η are well-typed according to Γ and define it inductively as follows.

$$
\vdash \emptyset, \emptyset : \mathsf{empty} \qquad \frac{\vdash \rho, \eta : \Gamma \quad v \in \mathcal{T}[\![A]\!]\eta}{\vdash \rho(x{:=}v), \eta : \mathsf{extend}(\Gamma, x, A)} \qquad \frac{\vdash \rho, \eta : \Gamma}{\vdash \rho, V\eta : \mathsf{tyExtend}(\Gamma)}
$$

5.4 Type Soundness

We prove type soundness for System F, that is, that well-typed programs cannot go wrong. Or more precisely, that if a program is well-typed and has type A, then its meaning (in a well-typed environment) is a subset of the meaning of its type A.

$$\begin{aligned}
\mathcal{T}[\![\texttt{int}]\!]\eta &= \mathbb{Z} \\
\mathcal{T}[\![i]\!]\eta &= \begin{cases} \text{cleanup}(\eta_i) & \text{if } i < |\eta| \\ \emptyset & \text{otherwise} \end{cases} \\
\mathcal{T}[\![A \to B]\!]\eta &= \{t \mid \forall (v_1, v_2') \in t.\, v_1 \in \mathcal{T}[\![A]\!]\eta \implies \exists v_2.\, v_2 \in \mathcal{T}[\![B]\!]\eta \wedge v_2' \sqsubseteq v_2\} \\
\mathcal{T}[\![\forall A]\!]\eta &= \{\text{thunk}(\text{some}(v)) \mid \forall V.\, v \in \mathcal{T}[\![A]\!]V\eta\} \cup \{\text{thunk}(\text{none})\}
\end{aligned}$$

$$\text{cleanup}(V) \equiv \{v \mid \exists v'.\, v' \in V \wedge v \sqsubseteq v' \wedge v \neq \text{wrong}\}$$

Fig. 10. A semantics of types for System F.

Theorem 4 (Semantic Soundness).
If $\Gamma \vdash e : A$ *and* $\vdash \rho, \eta : \Gamma$*, then* $\mathcal{E}[\![e]\!]\rho \subseteq \mathcal{T}[\![A]\!]\eta$*.*

We require four small lemmas. The first corresponds to Milner's Proposition 1 (Milner 1978).

Lemma 5.1 ($\mathcal{T}$ is downward closed). *If* $v \in \mathcal{T}[\![A]\!]\eta$ *and* $v' \sqsubseteq v$*, then* $v' \in \mathcal{T}[\![A]\!]\eta$*.*

Proof sketch. A straightforward induction on A □

Lemma 5.2 (wrong not in $\mathcal{T}$). *For any* A *and* η*,* wrong $\notin \mathcal{T}[\![A]\!]\eta$*.*

Proof sketch. A straightforward induction on A. □

Lemma 5.3. *If* $\vdash \rho : \Gamma$*, then* Γ *is a well-formed type environment.*

Proof sketch. This is proved by induction on the derivation of $\vdash \rho : \Gamma$. □

Lemma 5.4. $\mathcal{T}[\![A]\!](\eta_1\eta_3) = \mathcal{T}[\![\uparrow^{|\eta_2|}_{|\eta_1|}(A)]\!](\eta_1\eta_2\eta_3)$

Proof sketch. This lemma is proved by induction on A. □

Next we prove one or two lemmas for each language feature. Regarding term variables, we show that variable lookup is sound in a well-typed environment.

Lemma 5.5 (Variable Lookup). *If* $\vdash \rho, \eta : \Gamma$ *and* $x \in \text{dom}(\Gamma)$*, then* $\rho(x) \in \mathcal{T}[\![\text{lookup}(\Gamma, x)]\!]\eta$

Proof sketch. The proof is by induction on the derivation of $\vdash \rho, \eta : \Gamma$. The first two cases are straightforward but the third case requires some work and uses lemmas 5.3 and 5.4. □

The following lemma proves that function application is sound, similar to Milner's Proposition 2.

Lemma 5.6 (Application cannot go wrong). *If* $V \subseteq \mathcal{T}[\![A \to B]\!]\eta$ *and* $V' \subseteq \mathcal{T}[\![A]\!]\eta$*, then* apply$(V, V') \subseteq \mathcal{T}[\![B]\!]\eta$*.*

Proof sketch. The proof of this lemma is direct and uses lemmas 5.2 and 5.1. □

When it comes to polymorphism, our proof necessarily differs considerably from Milner's, as we must deal with first-class polymorphism. So instead of Milner's Proposition 4 (about type substitution), we instead have a Compositionality Lemma analogous to what you would find in a proof of Parametricity (Skorstengaard 2015). We need this lemma because the typing rule for type application is expressed in terms of type substitution ($e[B]$ has type $[0 \mapsto B]A$) but the meaning of $\forall A$ is expressed in terms of extending the environment η.

Lemma 5.7 (Compositionality).

$$\mathcal{T}[\![A]\!](\eta_1 V \eta_2) = \mathcal{T}[\![[|\eta_1| \mapsto B]A]\!](\eta_1\eta_2) \qquad \textit{where } V = \mathcal{T}[\![B]\!](\eta_1\eta_2).$$

Proof sketch. The proof is by induction on A. All of the cases were straightforward except for $A = \forall A'$. In that case we use the induction hypothesis to show that

$$\mathcal{T}[\![A']\!](V\eta_1 V_B \eta_2) = \mathcal{T}[\![([|V\eta_1| \mapsto \uparrow^1_0 (B)]\!]A'](V\eta_1\eta_2) \text{ where } V_B = \mathcal{T}[\![\uparrow^1_0 (B)]\!](V\eta_1\eta_2)$$

and use Lemma 5.4. □

The last lemma proves that iterate produces function tables of the appropriate type.

Lemma 5.8 (Iterate cannot go wrong). *If*

- $v \in \mathsf{iterate}(k, x, L, \rho)$ *and*
- *for any* v', $v' \in \mathcal{T}[\![A \to B]\!]\eta$ *implies* $L\ \rho(x{:=}v')\ \eta \subseteq \mathcal{T}[\![A{\to}B]\!]\eta$,

then $v \in \mathcal{T}[\![A{\to}B]\!]\eta$.

Proof sketch. This is straightforward to prove by induction on k. □

We proceed to the main theorem, that well-typed programs cannot go wrong.

Theorem 4 (Semantic Soundness).
If $\Gamma \vdash e : A$ *and* $\vdash \rho, \eta : \Gamma$*, then* $\mathcal{E}[\![e]\!]\rho \subseteq \mathcal{T}[\![A]\!]\eta$.

Proof sketch. The proof is by induction on the derivation of $\Gamma \vdash e : A$. The case for variables uses Lemmas 5.1 and 5.5. The case for application uses Lemma 5.6. The case for `fix` applies Lemma 5.8, using the induction hypothesis to establish the second premise of that lemma. The case for type application uses Lemma 5.7.

□

5.5 Comparison to Syntactic Type Safety

The predominant approach to proving type safety for small-step operational semantics is via *progress and preservation* (Harper 2012; Martin-Löf 1985; Pierce 2002; Wright and Felleisen 1994). The Progress Lemma says that a well-typed program can either reduce or it is a value, but it is never stuck, which would correspond to a runtime type error. The Preservation Lemma says that reduction preserves well-typedness. The strength of the syntactic approach is that the semantics does not need to explicitly talk about runtime type errors, but nevertheless they are ruled out by the Progress Lemma.

In comparison, the semantic soundness approach that we used in this section relies on using an explicit wrong value to distinguish between programs that diverge versus programs that encounter a runtime type error. The downside of this approach is that the author of the semantics could mixup wrong and zero (divergence) and the Semantic Soundness (Theorem 4) would still hold. However, simple auditing of the semantics can catch this kind of mistake. Also, as discussed in Section 2.1, we plan to investigate whether techniques such as step-indexing could be used to distinguish divergence from wrong.

5.6 From Type Soundness to Parametricity

An exciting direction for future research is to use the declarative semantics for proving Parametricity and using it to construct Free Theorems, replacing the frame models in the work of Wadler (1989). The idea would be to to adapt $\mathcal{T}$, our unary relation on values, into $\mathcal{V}$, a binary relation on values. We would define the following logical relation $\mathcal{R}$ in terms of $\mathcal{V}$ and the declarative semantics $\mathcal{E}$.

$$\mathcal{R}[\![A]\!]\eta = \{(e_1, e_2) \mid \exists v_1 v_2.\ v_1 \in \mathcal{E}[\![e_1]\!]\emptyset \wedge v_2 \in \mathcal{E}[\![e_1]\!]\emptyset \wedge (v_1, v_2) \in \mathcal{V}[\![A]\!]\eta\}$$

Then the Parametricity Theorem could be formulated as:

If $\Gamma \vdash e : A$ and $\vdash \rho, \eta : \Gamma$, then $(e, e) \in \mathcal{R}[\![A]\!]\eta$.

The proof would likely be similar to our proof of Semantic Soundness.

6 SEMANTICS OF MUTABLE REFERENCES

Our third case study addresses the question of whether declarative semantics are applicable to imperative languages, that is, languages with mutable state. We answer this question in the affirmative in the setting of a CBV simply-typed λ-calculus extended with mutable reference and pairs. The following are the types of this languages, where `ref` A is the type of a reference to a value of type A.

$$A, B \in \mathbb{T} \quad ::= \quad \texttt{int} \mid A \to B \mid A \times B \mid \texttt{ref}\, A \qquad \text{types}$$

The syntax of the language is defined by the following grammar and is standard, but to briefly review, `ref` e allocates a location in the store and initializes it with the result of e, returning the new address. The dereference operator $!e$ returns the value at the address that results from e. The assignment $e_1 := e_2$ updates the store at the address resulting from e_1 with the value from e_2.

$$\begin{array}{lll} e \in \mathbb{E} & ::= & n \mid e \oplus e \mid x \mid \lambda x{:}A.\, e \mid e\ e \mid (e_1, e_2) \mid \pi_i\, e \qquad \text{expressions} \\ & & \mid \texttt{ref}\, e \mid !e \mid e := e \end{array}$$

A pleasant finding regarding the declarative semantics for mutable references is that the definition of values does not change significantly. We can re-use function tables to represent the mutable store. So our only additions to the values are for addresses (which are natural numbers) and pairs.

$$\begin{array}{rcll} t & ::= & \{(v_1, v_1'), \ldots, (v_n, v_n')\} & \text{tables} \\ a & \in & \mathbb{N} & \text{addresses} \\ v \in \mathbb{V} & ::= & n \mid t \mid \mathsf{wrong} \mid (v, v) \mid a & \text{values} \end{array}$$

The ordering on values $\sqsubseteq$ is straightforward to extend to pairs and addresses:

$$n \sqsubseteq n \qquad \frac{t \subseteq t'}{t \sqsubseteq t'} \qquad \mathsf{wrong} \sqsubseteq \mathsf{wrong} \qquad \frac{v_1 \sqsubseteq v_1' \quad v_2 \sqsubseteq v_2'}{(v_1, v_2) \sqsubseteq (v_1', v_2')} \qquad a \sqsubseteq a$$

6.1 Adding State to the Monad

For some styles of semantics, such as big-step, the addition of mutable state would necessitate the explicit threading of the store through all of the evaluation rules. We shall instead combine a state monad (Wadler 1992) into the non-determinism and error monad that we used in Section 5 [5]. We define the monad type constructor M as follows and represent the store as a table.

$$\begin{aligned} \mathsf{store} &= \mathsf{fset}(\mathbb{V} \times \mathbb{V}) \\ M(\alpha) &= \mathsf{store} \to \mathsf{set}(\alpha \times \mathsf{store}) \end{aligned}$$

The monad operations bind and return become

$$\begin{aligned} \mathsf{return}\ v\, \mu &= \{(v, \mu)\} \\ \mathsf{bind}\ m\, f\, \mu_1 &= \{(v_2, \mu_3) \mid \exists v_1 \mu_2 .(v_1, \mu_2) \in m\, \mu_1 \wedge (v_2, \mu_3) \in f\, v_1\, \mu_2\} \end{aligned}$$

[5]We could use monad transformers to combine the state, error, and non-determinism monads (Liang et al. 1995), but we prefer to limit ourselves to one level of abstraction for the purposes of transparency of presentation.

The zero function remains unchanged (returns the empty set).

In several places in our non-deterministic interpreters (Figures 4 and 9) we use bind ($\longleftarrow$) with a right-hand side that is a set such as $\mathbb{N}$, $\mathbb{V}$, or t, and not the result of a sub-computation. This worked by coincidence because we were in the set monad. However, with the addition of state to the monad, this no longer works. We introduce a function named choose that injects a set into the monad by choosing an element of the set and pairing it with the current state.

$$\begin{aligned}\mathsf{choose} &: \mathsf{set}(\alpha) \to M(\alpha)\\ \mathsf{choose}\, S\, \mu &= \{(a, \mu) \mid a \in S\}\end{aligned}$$

Next we equip the monad with the operation get, that retrieves the current store, and the operation put, that overwrites the store.

$$\mathsf{get}\,\mu = \{(\mu, \mu)\} \qquad\qquad \mathsf{put}\,\mu_1\,\mu_2 = \{((), \mu_1)\}$$

6.2 Function Application with State

With the addition of mutable references, functions have side effects: they can change the store. To account for this, our function tables will not just map an input value to an output value, but will instead map an input value and store to an output value and store. The definition of the apply function is given below. We check that the current store μ_1 provides everything required by the input store μ, similar to how we check that the argument v_2 provides everything required by the input value v. If we have a match, then we install the output store μ' as the current store and return the output value v'. Otherwise we backtrack.

```
apply(V_1, V_2) =  $v_1 := V_1;\ v_2 := V_2;$
                   case $v_1$ of
                    $t \Rightarrow ((v,\mu),(v',\mu')) \longleftarrow \mathsf{choose}\,t;\ \mu_1 \longleftarrow \mathsf{get};$
                        if $v \sqsubseteq v_2 \wedge \mu \sqsubseteq \mu_1$ then $(\_ \longleftarrow \mathsf{put}\,\mu'; \mathsf{down}\ v')$ else zero
                   $|\ \_ \Rightarrow$ return wrong
```

6.3 Declarative Semantics with State

Figure 11 defines the declarative semantics of the CBV λ-calculus with mutable references. The cases for integers, arithmetic, and variables is the same as before, thanks to the monad. The λ abstraction case, however, changes according to the way state is handled by function application. When building the table t for the λ abstraction, we non-deterministically choose an input store and install it as the current store. We then apply the meaning of the body to obtain the output value v' and store μ'. The resulting table entry is $((v, \mu), (v', \mu'))$. Of course, the creation of a λ abstraction should not have side effects, so prior to building the table, we record the current state μ_1 and reinstate it afterwards. The cases for mutable references: allocation, dereference, and update, are all straightforward and what one would expect using a state and non-determinism monad.

7 RELATED WORK

Intersection Type Systems. The type system view of our declarative semantics (Section 2.4) is a variant of the intersection type system invented by Coppo et al. (1979) to study the untyped λ-calculus. They proved that their intersection type system was equivalent to the $P\omega$ model (Scott 1976) and to operational equivalence (Morris 1968). Subsequently, researchers studied numerous properties and variations of the intersection type system (Alessi et al. 2003, 2006; Coppo et al. 1984, 1987; Hindley 1982). Our mechanized proof of completeness with respect to operational semantics (Section 3.2) is based on a (non-mechanized) proof by Alessi et al. (2003).

$$
\begin{aligned}
\mathcal{E}[\![n]\!]\rho &= \text{return}\, n\\
\mathcal{E}[\![e_1 \oplus e_2]\!]\rho &= \;v_1 := \mathcal{E}[\![e_1]\!]\rho;\; v_2 := \mathcal{E}[\![e_2]\!]\rho;\\
&\quad\;\mathbf{case}\,(v_1, v_2)\,\mathbf{of}\,(n_1, n_2) \Rightarrow \text{return}\, n_1 \oplus n_2 \mid \_ \Rightarrow \text{return wrong}\\
\mathcal{E}[\![x]\!]\rho &= \text{down}\,\rho(x)\\
\mathcal{E}[\![\lambda x{:}A.\, e]\!]\rho &= \;\mu_1 \leftarrow \text{get}; k \leftarrow \text{choose}\,\mathbb{N}; vs \leftarrow \mathbb{V}^k;\\
&\quad\; t \leftarrow \text{map}\; vs \left(\lambda v. \begin{array}{l} \mu \leftarrow \text{choose fset}(\mathbb{V}\times\mathbb{V}); \_ \leftarrow \text{put}\,\mu;\\ v' := \mathcal{E}[\![e]\!]\rho(x{:=}v); \mu' \leftarrow \text{get}; \text{return}\,((v,\mu),(v',\mu'))\end{array}\right)\\
&\quad\; \_ \leftarrow \text{put}\,\mu_1;\; \text{return}\; t\\
\mathcal{E}[\![e_1\; e_2]\!]\rho &= \text{apply}(\mathcal{E}[\![e_1]\!]\rho, \mathcal{E}[\![e_2]\!]\rho)\\
\mathcal{E}[\![(e_1, e_2)]\!]\rho &= \;v_1 := \mathcal{E}[\![e_1]\!]\rho;\; v_2 := \mathcal{E}[\![e_2]\!]\rho;\; \text{return}\; (v_1, v_2)\\
\mathcal{E}[\![\pi_i\; e]\!]\rho &= v := \mathcal{E}[\![e]\!]\rho;\;\; \mathbf{case}\, v\, \mathbf{of}\, (v_1, v_2) \Rightarrow \text{return}\; v_i\\
&\qquad\qquad\qquad\quad \mid \_ \Rightarrow \text{return wrong}\\
\mathcal{E}[\![\texttt{ref}\; e]\!]\rho &= \;v := \mathcal{E}[\![e]\!]\rho; \mu \leftarrow \text{get}; a \leftarrow \text{choose}\,\mathbb{N};\\
&\quad\; \mathbf{if}\, a \notin \text{dom}(\mu)\, \mathbf{then}\, (\_ \leftarrow \text{put}\,(\{(a, v)\} \cup \mu); \text{return}\; a)\, \mathbf{else}\, \text{zero}\\
\mathcal{E}[\![!e]\!]\rho &= v := \mathcal{E}[\![e]\!]\rho;\;\; \mathbf{case}\, v\, \mathbf{of}\, a \Rightarrow\;\; \mu \leftarrow \text{get}; (a', v') \leftarrow \text{choose}\,\mu;\\
&\qquad\qquad\qquad\qquad\qquad\qquad\;\; \mathbf{if}\, a = a'\, \mathbf{then}\, \text{return}\; v'\, \mathbf{else}\, \text{zero}\\
&\qquad\qquad\qquad\quad \mid \_ \Rightarrow \text{return wrong}\\
\mathcal{E}[\![e_1 := e_2]\!]\rho &= \;v_1 := \mathcal{E}[\![e_1]\!]\rho;\; v_2 := \mathcal{E}[\![e_2]\!]\rho;\\
&\quad\; \mathbf{case}\, v_1\, \mathbf{of}\, a \Rightarrow \mu \leftarrow \text{get};\; \_ \leftarrow \text{put}\, \{(a, v_2)\} \cup (\mu - a); \text{return}\; a\\
&\quad\; \mid \_ \Rightarrow \text{return wrong}
\end{aligned}
$$

Fig. 11. The declarative semantics of mutable references.

By making subtle changes to the subtyping relation it is possible to capture alternate semantics (Alessi et al. 2004) such as call-by-value (Egidi et al. 1992; Ronchi Della Rocca and Paolini 2004) or lazy evaluation (Abramsky 1987). Barendregt et al. (2013) give a thorough survey of these type systems. Our $\top$ type corresponds to the type ν of Egidi et al. (1992) and Alessi et al. (2003). Our subtyping relation is rather minimal, omitting the usual rules for function subtyping and the distributive rule for intersections and function types. Our study of singleton integer types within an intersection type system appears to be a novel combination.

Intersection type systems have played a role in the full abstraction problem for the lazy λ-calculus, in the guise of domain logics (Abramsky 1987, 1990; Jeffrey 1993)

The problem of inhabitation for intersection type systems has seen recent progress (Dudenhefner and Rehof 2017) and applications to example-directed synthesis (Frankle et al. 2016).

Declarative Semantics. The model-theoretic semantics for logic programming languages (Emden and Kowalski 1976; Lloyd 1984) is declarative in the same sense as our semantics is declarative, it makes nontrivial use of existential quantifiers. A query (or goal) G is true in program P if there exists a substitution S such that $S(G)$ is a logical consequence of P. Unfortunately, the operational semantics for logic programs is incomplete with respect to this declarative semantics. Andrews (1992) tackled this problem by developing a proof-theoretic version of the declarative semantics that better matched the operational semantics. Of course, the declarative semantics presented in this paper differs from those for logic programming languages in that it gives meaning to functions.

Carlos et al. (1992) give operational, denotational (fixpoint), and declarative (model-theoretic) semantics for a conditional rewriting system, that is, a kind of lazy functional language but without λ abstraction. They show that the operational and declarative semantics are equivalent but that the denotational one is different.

Other Semantics. There are many other approaches to programming language semantics that we have not discussed, from axiomatic semantics (Floyd 1969; Hoare 1969) to games (Abramsky et al. 2000; Hyland and Ong 2000), event structures (Winskel 1987), and traces (Jeffrey and Rathke 2005; Reynolds 2004). The query-response way of thinking about functions that we discussed in Section 2.1 is reminiscent of game semantics. Our function tables are finitary versions of the tree models for SPCF (Cartwright et al. 1994; Cartwright and Felleisen 1992).

8 CONCLUSIONS AND FUTURE WORK

In this paper we present a semantics for an applied CBV λ-calculus that is compositional, extensional, and elementary. The semantics achieves compositionality by strategic uses of existential quantifiers. That is, the semantics is declarative and not operational. The semantics represents a λ abstraction with an infinite set of finite tables. We proved that this semantics is correct with respect to the operational semantics of the CBV λ-calculus and we present three case studies that begin to demonstrate that this semantics is useful. We leverage the compositionality of our semantics in a proof of correctness for a compiler optimization. We extend the semantics to handle parametric polymorphism and prove type soundness, i.e., well-typed programs cannot go wrong. Finally, we extend the semantics to handle mutable references, demonstrating that the approach scales to imperative programming languages.

Of course, we have just scratched the surface in investigating how well declarative semantics scales to full programming languages. We invite the reader to help us explore declarative semantics for objects, recursive types, dependent types, continuations, threads, shared memory, and low-level languages, to name just a few. Regarding applications, there is plenty to try regarding proofs of program correctness and compiler correctness. For your next programming languages project, give declarative semantics a try!

ACKNOWLEDGMENTS

I would like to thank Alan Jeffrey for his timely tweets regarding subsumption and the connection with domain logic that helped get this research off the ground. I thank Sam Tobin-Hochstadt for lots of great discussions about this work and thank Chung-chieh Shan for the help with monads. Also, a big thanks to those who gave feedback on drafts of this paper: Ronald Garcia, Daniel Friedman, Philip Wadler, and Larry Moss.

This material is based upon work supported by the National Science Foundation under Grant No. CCF-1518844. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Samson Abramsky. 1987. *Domain Theory and the Logic of Observable Properties.* Ph.D. Dissertation. University of London.

S. Abramsky. 1990. The Lazy Lambda Calculus. In *Research Topics in Functional Programming*. Addison-Welsey, Reading, MA, 65–116.

Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. 2000. Full abstraction for PCF. *Inf. Comput.* 163, 2 (2000), 409–470.

Peter Aczel. 1988. *Non-well-founded Sets.* Number 14 in CSLI Lecture Notes. Center for the Study of Language and Information.

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent Representation Independence. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, NY, USA, 340–353. https://doi.org/10.1145/1480881.1480925

Fabio Alessi, Franco Barbanera, and Mariangiola Dezani-Ciancaglini. 2003. Intersection Types and Computational Rules. *Electronic Notes in Theoretical Computer Science* 84 (2003), 45 – 59. https://doi.org/10.1016/S1571-0661(04)80843-0 WoLLIC'2003, 10th Workshop on Logic, Language, Information and Computation.

Fabio Alessi, Franco Barbanera, and Mariangiola Dezani-Ciancaglini. 2004. *Tailoring Filter Models*. Springer Berlin Heidelberg, Berlin, Heidelberg, 17–33. https://doi.org/10.1007/978-3-540-24849-1_2

Fabio Alessi, Franco Barbanera, and Mariangiola Dezani-Ciancaglini. 2006. Intersection Types and Lambda Models. *Theoretical Compututer Science* 355, 2 (2006), 108–126.

Nada Amin and Tiark Rompf. 2017. Type Soundness Proofs with Definitional Interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 666–679. https://doi.org/10.1145/3009837.3009866

J.H. Andrews. 1992. *Logic Programming: Operational Semantics and Proof Theory*. Cambridge University Press. https://books.google.com/books?id=ahq5LSrrO0AC

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized Metatheory for the Masses: The POPLmark Challenge. (May 2005).

Anindya Banerjee. 1997. A Modular, Polyvariant and Type-based Closure Analysis. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/258948.258951

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic* 48, 4 (001 12 1983), 931–940. https://doi.org/10.2307/2273659

Henk Barendregt, Wil Dekkers, and Richard Statman. 2013. *Lambda Calculus with Types*. Cambridge University Press. https://doi.org/10.1017/CBO9781139032636

Nick Benton, Andrew Kennedy, and Carsten Varming. 2009. *Some Domain Theory and Denotational Semantics in Coq*. Springer Berlin Heidelberg, Berlin, Heidelberg, 115–130. https://doi.org/10.1007/978-3-642-03359-9_10

Val Breazu-Tannen and Thierry Coquand. 1988. Extensional models for polymorphism. *Theoretical Computer Science* 59, 1 (1988), 85 – 114. https://doi.org/10.1016/0304-3975(88)90097-7

Kim B. Bruce, Albert R. Meyer, and John C. Mitchell. 1990. The semantics of second-order lambda calculus. *Information and Computation* 85, 1 (1990), 76 – 134. https://doi.org/10.1016/0890-5401(90)90044-I

Juan Carlos, González Moreno, Maria Teresa Hortalá González, and Mario Rodríguez Artalejo. 1992. *Denotational versus declarative semantics for functional programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 134–148. https://doi.org/10.1007/BFb0023763

Robert Cartwright, Pierre-Louis Curien, and Matthias Felleisen. 1994. Fully abstract semantics for observably sequential languages. *Inf. Comput.* 111, 2 (1994), 297–401.

Robert Cartwright and Matthias Felleisen. 1992. Observable sequentiality and full abstraction. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 328–342.

Arthur Charguéraud. 2013. Pretty-Big-Step Semantics. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems (ESOP'13)*. Springer-Verlag, Berlin, Heidelberg, 41–60. https://doi.org/10.1007/978-3-642-37036-6_3

Alonzo Church. 1932. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics* 33, 2 (1932), pp. 346–366. http://www.jstor.org/stable/1968337

M. Coppo, M. Dezani-Ciancaglini, F. Honsell, and G. Longo. 1984. Extended Type Structures and Filter Lambda Models. In *Logic Colloquium '82*, G. Longo G. Lolli and A. Marcja (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 112. Elsevier, 241 – 262. https://doi.org/10.1016/S0049-237X(08)71819-6

M. Coppo, M. Dezani-Ciancaglini, and P. Salle'. 1979. *Functional characterization of some semantic equalities inside λ-calculus*. Springer Berlin Heidelberg, Berlin, Heidelberg, 133–146. https://doi.org/10.1007/3-540-09510-1_11

M. Coppo, M. Dezani-Ciancaglini, and M. Zacchi. 1987. Type theories, normal forms, and Dfk-lambda-models. *Information and Computation* 72, 2 (1987), 85 – 116. https://doi.org/10.1016/0890-5401(87)90042-3

Thierry Coquand, Carl Gunter, and Glynn Winskel. 1989. Domain theoretic models of polymorphism. *Information and Computation* 81, 2 (1989), 123 – 167. https://doi.org/10.1016/0890-5401(89)90068-0

H. B. Curry and R. Feys. 1958. *Combinatory Logic,*. North-Holland Publishing Co., Amsterdam.

Robert Dockins. 2014. Formalized, Effective Domain Theory in Coq. In *Interactive Theorem Proving (ITP)*.

Andrej Dudenhefner and Jakob Rehof. 2017. Intersection Type Calculi of Bounded Dimension. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 653–665. https://doi.org/10.1145/3009837.3009862

Lavinia Egidi, Furio Honsell, and Simona Ronchi Della Rocca. 1992. Operational, Denotational and Logical Descriptions: A Case Study. *Fundam. Inf.* 16, 2 (Feb. 1992), 149–169. http://dl.acm.org/citation.cfm?id=161643.161646

M. H. Van Emden and R. A. Kowalski. 1976. The Semantics of Predicate Logic as a Programming Language. *J. ACM* 23, 4 (1976), 733–742.

Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. 1987. A syntactic theory of sequential control. *Theoretical Computer Science* 52, 3 (1987), 205 – 237. https://doi.org/10.1016/0304-3975(87)90109-5

R. W. Floyd. 1969. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics*, Vol. 19. 19–32.

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-directed Synthesis: A Type-theoretic Interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16).* ACM, New York, NY, USA, 802–815. https://doi.org/10.1145/2837614.2837629

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic Subtyping: Dealing Set-theoretically with Function, Union, Intersection, and Negation Types. *J. ACM* 55, 4, Article 19 (Sept. 2008), 64 pages.

Jean-Yves Girard. 1972. *Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur.* Ph.D. Dissertation. Paris, France.

Carl A. Gunter. 1992. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, Cambridge, MA, USA.

Professor Robert Harper. 2012. *Practical Foundations for Programming Languages.* Cambridge University Press, New York, NY, USA.

J. R. Hindley. 1982. *The simple semantics for Coppo-Dezani-Sallé types.* Springer Berlin Heidelberg, Berlin, Heidelberg, 212–226. https://doi.org/10.1007/3-540-11494-7_15

C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

Chung-Kil Hur and Derek Dreyer. 2011. A Kripke Logical Relation Between ML and Assembly. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11).* ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1926385.1926402

J. M. E. Hyland and C.-H. L. Ong. 2000. On full abstraction for PCF: I, II, and III. *Inf. Comput.* 163, 2 (2000), 285–408.

Suresh Jagannathan and Andrew Wright. 1996. Flow-directed Inlining. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI '96).* ACM, New York, NY, USA, 193–205. https://doi.org/10.1145/231379.231417

Alan Jeffrey. 1993. *A fully abstract semantics for concurrent graph reduction.* Technical Report 12/93. University of Sussex.

A. S. A. Jeffrey and J. Rathke. 2005. Java Jr.: Fully Abstract Trace Semantics for a Core Java Language. In *Proc. European Symposium on Programming.*

Gilles Kahn. 1987. Natural Semantics. In *Symposium on Theoretical Aspects of Computer Science.* 22–39.

Neelakantan Krishnaswami. 2013. What Declarative Languages Are. (2013). http://semantic-domain.blogspot.com/2013/07/what-declarative-languages-are.html

P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (1964), 308–320.

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad transformers and modular interpreters. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM Press, New York, NY, USA, 333–343.

John Wylie Lloyd. 1984. *Declarative Semantics.* Springer Berlin Heidelberg, Berlin, Heidelberg, 1–34. https://doi.org/10.1007/978-3-642-96826-6_1

P. Martin-Löf. 1985. Constructive Mathematics and Computer Programming. In *Proc. Of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 167–184. http://dl.acm.org/citation.cfm?id=3721.3731

John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM* 3, 4 (1960), 184–195.

Nancy Jean McCracken. 1979. *An Investigation of a Programming Language with a Polymorphic Type Structure.* Ph.D. Dissertation. Syracuse, NY, USA. AAI7925584.

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375.

Robin Milner. 1995. *Communication and Concurrency.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK.

James H. Morris. 1968. *Lambda-calculus Models of Programming Languages.* Ph.D. Dissertation. MIT, Cambridge, MA, USA.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2007. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.* LNCS, Vol. 2283. Springer.

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632.* Springer-Verlag New York, Inc., New York, NY, USA, 589–615. https://doi.org/10.1007/978-3-662-49498-1_23

Benjamin C. Pierce. 2002. *Types and Programming Languages.* MIT Press.

Andrew M. Pitts. 1987. Polymorphism is Set Theoretic, Constructively. In *Category Theory and Computer Science*. Springer-Verlag, London, UK, UK, 12–39. http://dl.acm.org/citation.cfm?id=648331.755423

G.D. Plotkin. 1973. *Lambda-definability and logical relations.* Technical Report SAI-RM-4. University of Edinburgh, School of Artificial Intelligence.

G.D. Plotkin. 1981. *A Structural Approach to Operational Semantics.* Technical Report DAIMI FN-19. Computer Science Dept., Aarhus Universitet. https://books.google.com/books?id=tt7sjgEACAAJ

Gordon D. Plotkin. 1993. Set-theoretical and other elementary models of the -calculus. *Theoretical Computer Science* 121, 1 (1993), 351 – 409. https://doi.org/10.1016/0304-3975(93)90094-A

Gordon D Plotkin. 2004. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming* 60 (2004), 3 – 15. https://doi.org/10.1016/j.jlap.2004.03.009 Structural Operational Semantics.

John C. Reynolds. 1974. Towards a Theory of Type Structure. In *Programming Symposium (LNCS)*, Vol. 19. Springer-Verlag, 408–425.

John C. Reynolds. 1984. *Polymorphism is not set-theoretic.* Springer Berlin Heidelberg, Berlin, Heidelberg, 145–156. https://doi.org/10.1007/3-540-13346-1_7

John C. Reynolds. 2004. Toward a Grainless Semantics for Shared-Variable Concurrency. In *Proceedings of the 24th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2004).*

Simona Ronchi Della Rocca and Luca Paolini. 2004. *The Parametric Lambda Calculus.* Springer.

David A. Schmidt. 2003. Programming Language Semantics. In *Encyclopedia of Computer Science.* John Wiley and Sons Ltd., Chichester, UK, 1463–1466. http://dl.acm.org/citation.cfm?id=1074100.1074733

Dana Scott. 1970. *Outline of a Mathematical Theory of Computation.* Technical Report PRG-2. Oxford University.

Dana Scott. 1976. Data Types as Lattices. *SIAM J. Comput.* 5, 3 (1976), 522–587.

Dana Scott and Christopher Strachey. 1971. *Toward a mathematical semantics for computer languages.* Technical Report PRG-6. Oxford Programming Research Group.

Dana S. Scott. 1982. *Domains for denotational semantics.* Springer Berlin Heidelberg, Berlin, Heidelberg, 577–610. https://doi.org/10.1007/BFb0012801

Dana S. Scott. 1993. A type-theoretic alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science* 121, 411-440 (1993).

Jeremy G. Siek. 2012. Big-step, diverging or stuck? (July 2012). http://siek.blogspot.com/2012/07/big-step-diverging-or-stuck.html

Lau Skorstengaard. 2015. *An Introduction to Logical Relations.* Technical Report. Aarhus University.

R. Statman. 1985. Logical relations and the typed -calculus. *Information and Control* 65, 2–3 (1985), 85 – 97. https://doi.org/10.1016/S0019-9958(85)80001-2

Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08).* ACM, New York, NY, USA, 17–27. https://doi.org/10.1145/1328438.1328444

Franklyn Turbak, Allyn Dimock, Robert Muller, and J. B. Wells. 1997. Compiling with Polymorphic and Polyvariant Flow Types. In *Workshop on Types in Compilation.*

Oscar Waddell and R. Kent Dybvig. 1997. Fast and Effective Procedure Inlining. In *Proceedings of the 4th International Symposium on Static Analysis (SAS '97).* Springer-Verlag, London, UK, 35–52.

Philip Wadler. 1989. Theorems for free!. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture.* ACM, 347–359.

Philip Wadler. 1992. The essence of functional programming. In *Symposium on Principles of Programming Languages.*

Glynn Winskel. 1987. Event Structures. In *Advances in Petri Nets 1986 (LNCS).*

Glynn Winskel. 1993. *The Formal Semantics of Programming Languages.* MIT Press.

Andrew K. Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and Computation* 115, 1 (1994), 38–94.