J. Functional Programming **11** (3): 263–317, May 2001. Printed in the United Kingdom

*From Polyvariant flow information to intersection and union types**

JENS PALSBERG and CHRISTINA PAVLOPOULOU
Department of Computer Science, Purdue University, W. Lafayette, IN 47907, USA
(*e-mail:* palsberg@cs.purdue.edu)

Abstract

Many polyvariant program analyses have been studied in the 1990s, including k-CFA, polymorphic splitting, and the cartesian product algorithm. The idea of polyvariance is to analyze functions more than once and thereby obtain better precision for each call site. In this paper we present an equivalence theorem which relates a co-inductively-defined family of polyvariant flow analyses and a standard type system. The proof embodies a way of understanding polyvariant flow information in terms of union and intersection types, and, conversely, a way of understanding union and intersection types in terms of polyvariant flow information. We use the theorem as basis for a new flow-type system in the spirit of the λ^{CIL}-calculus of Wells, Dimock, Muller and Turbak, in which types are annotated with flow information. A flow-type system is useful as an interface between a flow-analysis algorithm and a program optimizer. Derived systematically via our equivalence theorem, our flow-type system should be a good interface to the family of polyvariant analyses that we study.

Capsule Review

Connections between monovariant flow analyses and type systems are well known. This paper presents the first formal connection between polyvariant flow analyses and a type system with intersection and union types. Intuitively, intersection types model the analysis of functions in multiple contexts, while union types model sets of abstract closures.

The paper introduces F-analyses, a novel flow framework that can express flow analyses with argument-based polyvariance, such as Agesen's cartesian product analysis and the polyvariant analysis of Schmidt, as well as the monovariant 0-CFA analysis. The key result of the paper is that a program is safety checkable in an F-analysis if and only if it is typable with intersection and union types. The correspondence is given by translations between F-analyses and type derivations.

The results presented here are especially important in the context of type-directed compilation, where it is desirable to encode the results of static analyses in the type system of a typed intermediate language. The paper concludes with the design of a flow-type system that can be used as the basis of a typed intermediate language.

* A preliminary version of this paper appeared in the *Proceedings of POPL'98: 25th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 197–208, San Diego, CA, January 1998.

1 Introduction

1.1 Background

Flow analysis of higher-order programs is done for a variety of reasons, including: closure conversion (Wand and Steckler, 1994), binding-time analysis (Bondorf, 1991), optimizing strict functional programs (Jagannathan and Wright, 1995), optimizing non-strict functional programs (Faxén, 1995), optimizing object-oriented programs (Pande and Ryder, 1996), optimizing concurrent programs (Plevyak *et al.*, 1995), safety checking (Graver and Johnson, 1990; Palsberg and Schwartzbach, 1995), and detecting uncaught exceptions (Yi, 1994). A basic, often-seen form of flow analyses can be done in $O(n^3)$ time where n is the size of the program. This so-called *monovariant* form of analysis can be varied in minor ways without changing the time complexity (Heintze and McAllester, 1997b), and for simplicity we will refer to all of these variations as 0-CFA. (CFA is an abbreviation introduced by Shivers (1991); it stands for 'control-flow analysis'.) A common observation is that 0-CFA is sometimes rather imprecise, resulting in, for example, little or no optimization. The key property of 0-CFA is that each function is analyzed just once (or not at all, if the analysis is demand-driven).

The idea of *polyvariance* is to analyze functions more than once and thereby obtain better precision for each call site. Polyvariant analysis was pioneered by Sharir and Pnueli (1981), and Jones and Muchnick (1982). In the 1990s the study of polyvariant analysis has been intensive. Well known are the k-CFA of Shivers (1991), the poly-k-CFA of Jagannathan and Weeks (1995), the polymorphic splitting of Jagannathan and Wright (1995), and the cartesian product algorithm of Agesen (1995a; 1995b). A particularly simple polyvariant analysis was presented by Schmidt (1995). Frameworks for defining polyvariant analyses have been presented by Stefanescu and Zhou (1994), Jagannathan and Weeks (1995), and Nielson and Nielson (1997). Successful applications of polyvariant analysis include the optimizing compilers of Emami, Ghiya, and Hendren (1994), Grove, DeFouw, Dean, and Chambers (1997), the partial evaluator of Consel (1993), and the application extractor of Agesen and Ungar (1994).

Is polyvariance related to polymorphism? This question is becoming increasingly important for the many recent efforts to integrate flow analysis and type systems, as pioneered by Tang and Jouvelot (1994), Heintze (1995), Banerjee (1997), and Wells, Dimock, Muller and Turbak (1997; 1997; 1997). This line of work builds on earlier ideas on integrating strictness information and type systems, as first done by Kuo and Mishra (1989) and later by Wright (1991), Amtoft (1993), and others. Benefits of integrating flow analysis and type systems may include: easy correctness proofs (Heintze, 1995), faster flow analysis without sacrificing precision (Heintze and McAllester, 1997a), a definition of a both sound and complete flow analysis (Mossin, 1997), and a simplified compiler structure (Wells *et al.*, 1997). Intuitively, polyvariant analysis is closer to intersection types (Coppo *et al.*, 1980; Hindley, 1991; Jim, 1996b) and union types (Pierce, 1991; Barbanera *et al.*, 1995) than to universal and existential quantifiers (Milner, 1978), as observed by Banerjee (1997), Wells, Dimock, Muller and Turbak (1997), and others. The insight is that

- 'analyzing a function a number of times' can be modeled by an intersection type; and
- 'a set of abstract values' can be modeled by a union type.

In simple cases, these effects can also be achieved via universal quantifiers, for example, in binding-time analysis (Henglein and Mossin, 1994). In general, polyvariance seems to be a concept distinct from universal and existential quantification. Further results on using universal quantifiers to achieve a form of polyvariance have been presented by Jagannathan, Wright and Weeks (1997).

Our goal is a foundation for designing and understanding combinations of flow analyses and type systems. This leads us to the following question:

Question: How does flow analysis relate to type systems?

Let us first examine the key differences between flow analyses and type systems. Both may be understood as abstract interpretations, and the distinction between them lies mostly in how they are usually formulated:

Type system	*Flow analysis*
finitary	may be infinitary
may analyze all code	may avoid analyzing dead code
defined inductively	may be defined co-inductively
may reject some programs	works for all programs

A type system is usually defined using type rules. The rules inductively define a set of valid type judgments. In the derivation of a valid type judgment for a program, usually only finitely many types will be involved, and usually all parts of the program will be analyzed. Moreover, for some type systems, there will be some 'not type correct' programs for which no valid type judgments exists. In contrast, it has recently been argued that flow analyses can in a natural way be defined co-inductively (Nielson and Nielson, 1997). The flow analysis of a program may involve infinitely many abstract values, and often a form of reachability is built in which avoids the analysis of dead code. Moreover, there will usually be valid flow judgments for all programs.

When we attempt to relate flow analyses and type systems, we must find a way of handling the four basic differences. In this paper we choose to do that by letting the type systems 'have it their way'. Following Nielson and Nielson (1997), we give a co-inductive definition of a family of flow analyses, and then we restrict attention to flow judgments that can be proved using finitely many abstract values, and which analyze all parts of a program. Moreover, we enforce safety checks, e.g. checks that the flow set for the operator part of a function application actually only contains abstract closures, thereby making the augmented flow analysis accept and reject programs much like a type checker. With the safety checks in place, we present a type system which accepts exactly the safety-checkable programs. A similar agenda has been carried out for 0-CFA by Palsberg and O'Keefe (1995), and the related type

system turned out to be the one of Amadio and Cardelli (1993) with subtyping and recursive types. Three more such results for restrictions of 0-CFA have later been presented by Heintze (1995). (One of Heintze's results was not completely correct; see the paper by Palsberg (1998) for a correct version of that result.) The Amadio-Cardelli type system, in turn, is equivalent to a form of constrained types (Mitchell, 1984; Mitchell, 1991; Aiken and Wimmers, 1993; Eifrig *et al.*, 1995b; Eifrig *et al.*, 1995a), as shown by Palsberg and Smith (1996). In this paper, we address the above question for a family of polyvariant flow analyses.

Let us next examine three of the main approaches to polyvariant flow analysis. The best known approach may be that of using *call-strings*, that is, finite approximations of the dynamic call chain, to differentiate calling contexts. Examples of such analyses include k-CFA (Shivers, 1991) and poly-k-CFA (Jagannathan and Weeks, 1995). A more recent approach to polyvariant analysis is the polymorphic splitting of Jagannathan and Wright (1995), which for a let-bound expression does a separate analysis for each occurrence of the let-variable. Finally, the cartesian product algorithm of Agesen (1995a; 1995b) will, for a given function, perform a separate analysis for each possible *static* environment of the body.

Nielson and Nielson (1997) model the call-string and polymorphic splitting approaches using so-called mementa (sequences of labels and expressions) and mementa environments. We will focus on modeling the style of polyvariance of the cartesian product algorithm. Our approach is to model a function by an abstract closure which consists of the function and an abstract environment. The call-string-oriented analyses, like k-CFA, cannot be expressed in our framework. In addition to the cartesian product algorithm, our framework can also model a flow analysis of Schmidt (1995), and as a simple case also 0-CFA.

1.2 Our family of flow analyses

Our family of polyvariant flow analyses is based on the notion of *cover*, as explained in the following.

Suppose there is a call site where we want to abstractly apply a λ-abstraction $\lambda x.e$ to a flow set s. If $s \subseteq s'$, then we may choose to analyze the body e in an environment where x is bound to s'. If we make the choice s' for all call sites, then we effectively do just one analysis of the body e, no matter how many other call sites there may be where $\lambda x.e$ can be invoked. This is the approach of 0-CFA-style flow analysis (Palsberg, 1995), which is a form of monovariant flow analysis.

In the situation just described, binding x to s' is a rather conservative choice. A polyvariant analysis may choose, at a given call site, to bind x to the flow set s for the actual argument. If we make this choice for all call sites, then we may get to analyze the body e in a number of different environments, depending on the number of different flow sets for the actual arguments. Notice that this number may be less than the number of call sites where $\lambda x.e$ can be invoked, because two call sites may have arguments with the same flow set. This is essentially the approach of the relational closure analysis of Schmidt Section 12. This style of analysis was later rediscovered by Grove *et al.* (1997) who called it Simple Class Sets (SCS).

We can do a more fine-grained polyvariant analysis by breaking up the flow set for the actual argument into singleton sets. The idea is to do an analysis of the body e for each singleton set, and then take the union of the results. If we make this choice for all call sites, then we may get to analyze the body e in a number of different environments, depending on the number of different values in the flow sets for the actual arguments. This is the approach of the cartesian product algorithm of Agesen (1995a, 1995b).

To illustrate the differences among 0-CFA, Schmidt's analysis, and Agesen's analysis, suppose we have a program E, abstract values a_1, a_2, a_3, and two call sites occurring in E. Suppose also that at each of the call sites we can invoke the λ-abstraction $\lambda x.e$, and that the flow sets for the actual arguments are $\{a_1, a_2\}$ and $\{a_2, a_3\}$, respectively. With 0-CFA, we will do just one analysis of the body e in an environment where x is bound to $\{a_1, a_2\} \cup \{a_2, a_3\} = \{a_1, a_2, a_3\}$. With Schmidt's analysis, we will do two analyses of the body e: one where x is bound to $\{a_1, a_2\}$, and one where x is bound to $\{a_2, a_3\}$. With Agesen's analysis, we will do three analyses of the body e: one where x is bound to $\{a_1\}$, one where x is bound to $\{a_2\}$, and one where x is bound to $\{a_3\}$.

In section 4, we describe a family of polyvariant flow analyses which subsumes all of the above. The idea is that at a given call site, we will for each function that can be invoked choose a *cover* of the flow set for the actual argument, see figure 1. A cover is a set of abstract-value sets. The body of the function will then, at that call site, be analyzed one time for each element of the cover. The analysis result for the call site is the *union* of the results for the individual analyses. We can now summarize the ideas of 0-CFA, Schmidt's analysis, and Agesen's analysis in terms of how they cover the flow set for an actual argument:

- **0-CFA.** For a given function, the cover consists of a single set, see the illustration in figure 2. That set depends upon which function we consider. For a given function, the set is the union of the flow sets for the actual arguments at the call sites where the function can be applied.
- **Schmidt's analysis.** The cover of a set of abstract values s is $\{s\}$, see the illustration in Figure 3.
- **Agesen's analysis.** The cover of a set of abstract values s is $\{\ \{a\}\ \mid\ a \in s\ \}$, see the illustration in figure 4. Notice that if $s = \emptyset$, then the cover is $\emptyset$.

We will define the family of flow analyses such that covers can be chosen independently at different call sites. Thus, we do not require a uniform covering strategy. This means that our definition captures adaptive styles of flow analysis, in the spirit of Plevyak *et al.* (1995), where the degree of polyvariance is determined during the analysis.

Given a program E, we will define a function F which maps sets of flow judgments to sets of flow judgments. The set of valid flow judgments is then defined to be the greatest fixed point of F, and an F-analysis of E is defined to be a prefixed point R of F (that is, $R \subseteq F(R)$) which contains a flow judgment for E. Among the analyses captured in this way are those of Agesen, Schmidt, and as a simple case also 0-CFA. This yields the first formalization of an analysis in the style of Agesen.

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

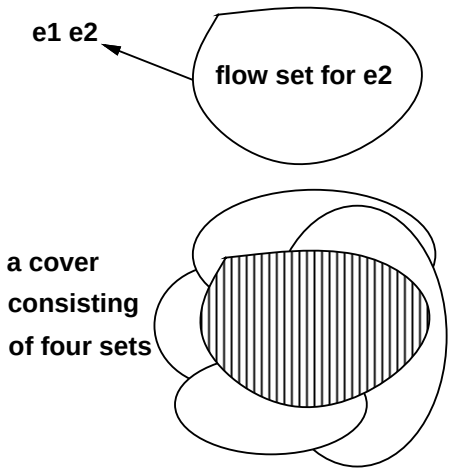

Fig. 1. Cover the argument!

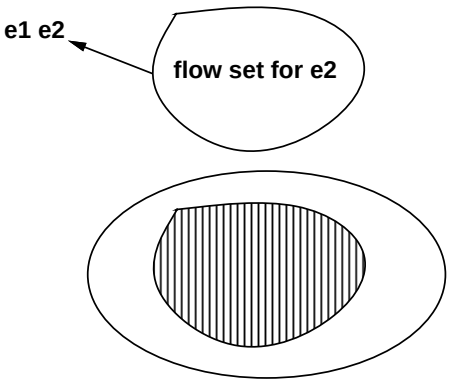

Fig. 2. 0-CFA.

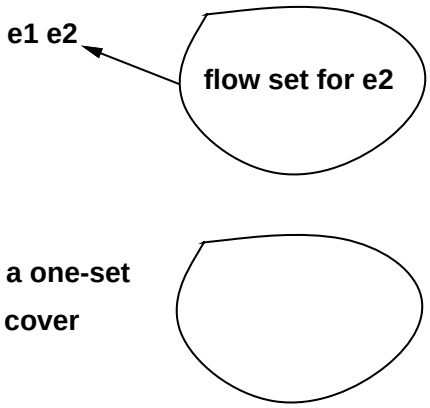

Fig. 3. Schmidt's analysis.

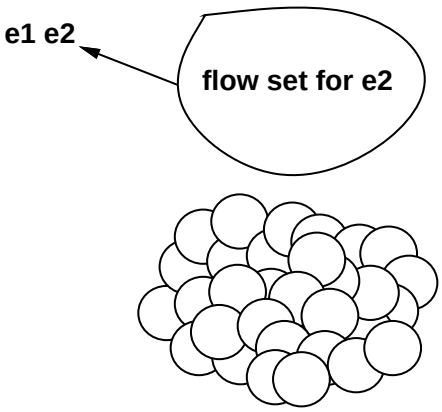

Fig. 4. Agesen's analysis.

One of the advantages of our definition is that it is flexible enough to accommodate changes to the covering strategy. For example, Agesen in his thesis (1995b) begins with explaining that his covering strategy is to use singleton sets. Later, he observes that this leads to an uncomputable analysis, and then he modifies the covering strategy to make the analysis computable. Our definition of an F-analysis is free from algorithmic concerns. In general, the set of valid flow judgments for a program is not decidable.

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

1.3 Our result

We present an equivalence theorem which relates a co-inductively-defined family of polyvariant flow analyses and a standard type system. The proof embodies a way of understanding polyvariant flow information in terms of union and intersection types, and conversely, a way of understanding union and intersection types in terms of polyvariant flow information. We use the theorem as basis for a new flow-type system in the spirit of the λ^{CIL}-calculus of Wells, Dimock, Muller and Turbak, in which types are annotated with flow information. A flow-type system is useful as an interface between a flow-analysis algorithm and a program optimizer. Derived systematically via our equivalence theorem, our flow-type system should be a good interface to the family of polyvariant analyses that we study.

Specifically, we prove that a program can be safety-checked by an analysis which is finitary (uses finitely many abstract values) and analyzes all parts of the program if and only if the program can be typed in a type system with intersection types, union types, subtyping, and recursive types.

To map flows to types, we (1) extract an equation system from the flow information, (2) solve the equation system, and (3) build a type derivation. The condition that the employed set of abstract values is finite ensures that the equation system is finite. We map types to flows in a similar way.

In slogan form, our result reads:

Polyvariance = Intersection Types + Union Types + Subtyping + Recursive Types.

The slogan should be taken with a grain of salt: our notion of polyvariance only covers some of the known approaches to polyvariance.

1.4 On proving correctness

We will define a language and equip it with a type system, a family of flow analyses, and a combined flow-type system. In this section we will discuss our choice of semantics for the language and its impact on proofs of correctness.

For each of (1) type systems, (2) flow analyses, and (3) flow-type systems it is of interest to prove *type/flow preservation*, that is, type/flow information is still valid after a number of computation steps. Moreover, for type systems and flow-type systems we want to establish *type soundness*, that is, a typable program cannot go wrong. For a flow analysis, we want to establish *flow soundness*, e.g. if a program evaluates to a function, then that function is represented in the flow information for the program. Usually, type/flow preservation can be used as a lemma for proving type/flow soundness.

When attempting to prove the listed properties, the key question is: what is the style of semantics for the language? We will discuss three of the possible choices: (1) small-step operational semantics with syntactic substitution, (2) small-step operational semantics with environments, and (3) big-step operational semantics with environments.

Small-step operational semantics with syntactic substitution. This is the style of semantics used in, for example, Barendregt (1981). It is convenient for proving type preservation and type soundness, both for type systems and flow-type systems, as will be exemplified in sections 3 and 7. It has also been used to prove flow preservation for a 0-CFA-style flow analysis (Palsberg, 1995). Unfortunately, it is problematic to use this style of semantics to prove flow preservation for the polyvariant flow analysis we study here. As will be exemplified in section 4, our family of flow analyses does *not* have the flow preservation property with respect to a standard small-step semantics with syntactic substitution. We know of no way of changing the definition of the polyvariant flow analysis such that flow preservation can be proved with respect to this style of semantics.

Small-step operational semantics with environments. This is also known as Plotkin-style operational semantics (Plotkin, 1981). It has been used by Nielson and Nielson (1997) to prove flow preservation for a parameterized polyvariant flow analysis. Can this style of semantics also be used to prove flow preservation for our analysis? We think the answer may be 'yes', but we have not pursued it. The reason is that it seems problematic to use this style of semantics to prove type preservation for our type system.

Big-step operational semantics with environments. This is also known as natural semantics (Kahn, 1987; Despeyroux, 1986). In contrast to the two styles of semantics discussed above, it cannot handle nonterminating programs. It has been used by Palsberg and Schwartzbach (1995) and by Schmidt (1995) to prove correctness theorems for some flow analyses. Milner and Tofte (1991) showed how to use it to prove type soundness for a type system. Their proof uses co-induction. We have not tried to use this style of semantics because we want to be able to handle nonterminating programs.

In summary, each of the three styles of semantics are problematic for our purposes. One alternative approach would be to have two semantics, and then (1) use one semantics for proving type preservation, (2) use another semantics for proving flow preservation, and (3) prove that the two semantics agree. Although this may be doable, we will not pursue it here.

We have chosen to use a small-step operational semantics with syntactic substitution. With respect to that semantics we will prove type preservation and type soundness for the type system (section 3) and the flow-type system (section 7). For the pure flow analysis, we will settle for a more coarse-grained and less powerful correctness result. This correctness result will be obtained via the equivalence with the type system (section 5) in the following way. For a program which is safety-checkable with a finitary polyvariant analysis which analyzes all parts of the program, we have that the program is typable in our type system. Since we have type soundness for the type system, it follows that the program cannot go wrong.

1.5 Open problems

Among the future work and open problems are:

- Define a semantics, a type system, and a polyvariant flow analysis in such a way that type/flow preservation and type/flow soundness all can be proved.
- Implement and experiment with the translation from flows to types.
- Provide a type inference algorithm for a large subset of our type system, perhaps a 'rank 2' fragment in the spirit of Jim (1996b). Our type system with intersection and union types is more generous than usually found in papers on type inference with intersection types (Coppo and Giannini, 1992; van Bakel, 1991; Jim, 1996a; Banerjee, 1997).
- Prove a principal flow property for our family of flow analyses, in the spirit of the result by Nielson and Nielson (1997).
- Extend the results to a combination of our framework and the framework of Nielson and Nielson.
- Prove an equivalence which is based on that the flow analyses 'have it their way', to as large extent as possible. For example, this entails working with a type system which avoids type checking dead code.
- Obtain our type rules systematically using the method of Cousot (1997).

Paper outline In the remainder of this section we illustrate our result by an example. In section 2 we define an example language, in section 3 we define our type system, in section 4 we present our family of polyvariant flow analyses, and in section 5 we prove our equivalence result. In section 6 we illustrate how different flow analyses lead to different typings, and in section 7 we define our flow-type system.

1.6 Example

The running example of this paper is the following program:

$$E = (\lambda f.\mathsf{succ}\ ((f f)\mathsf{0}))\ (\mathsf{if0}\ c\ (\lambda x.x)\ (\lambda y.\lambda z.z)).$$

This example is chosen because it requires a rather powerful polyvariant analysis to produce better flow information than 0-CFA. We assume that the condition c of the if0-expression does not cause any run-time error. If c terminates and one of the branches is passed to $\lambda f.\mathsf{succ}\ ((f f)\mathsf{0})$, then the result of $(f f)$ will be the identity function, and the result of evaluating the whole body will be succ 0. No run-time errors!

Notice also that a safety check based on 0-CFA fails. The 0-CFA flow information for the if0-expression is $s = \{\ \lambda x.x, \lambda y.\lambda z.z\ \}$. The flow information for $(f f)$ will now contain all possible results of applying an element of s to an element of s. There are 2×2 combinations. The result is $\{\ \lambda x.x, \lambda z.z, \lambda y.\lambda z.z\ \}$. When we then flow analyze $(f f)\mathsf{0}$, we get the flow information $\{\ \mathsf{Int}, \lambda z.z\ \}$. The safety check for succ $((f f)\mathsf{0})$ requires that the flow information for the argument of succ is a subset of $\{\mathsf{Int}\}$ so the safety check fails. This implies that the program is not typable in the type system with subtyping and recursive types of Amadio and Cardelli (1993), using the equivalence of Palsberg and O'Keefe (1995).

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

A safety check based on Schmidt's analysis also fails, for essentially the same reason that a safety check based on 0-CFA fails.

The problems encountered above during type checking and safety checking are similar. The types/flows for $\lambda x.x$ and $\lambda y.\lambda z.z$ are combined, and later we cannot get sufficiently precise information about the result of (ff). Agesen's polyvariant analysis improves the situation by enabling separate analyses of two copies of $\lambda f.\mathsf{succ}\ ((ff)\mathsf{0})$, one for each element of $\{\ \lambda x.x, \lambda y.\lambda z.z\ \}$. For both copies, the safety check succeeds, so the conclusion is that E does not cause any run-time error. In sections 4 and 5 we give full details of applying an Agesen-style analysis to the example program, and of mapping the flows to types. Here, we outline how the types obtained that way do indeed yield a type derivation for the program.

The effect of polyvariant analysis can be obtained in a type system by a combination of intersection types, union types, subtyping and recursive types. For the program E,

1. we check that $\lambda x.x$ has some type σ, and that $\lambda y.\lambda z.z$ has some type τ, and then we get that the if0-expression has the union type $(\sigma \vee \tau)$, and
2. we check that $\lambda f.\mathsf{succ}\ ((ff)\mathsf{0})$ has the types $\sigma \to \mathsf{Int}$ and $\tau \to \mathsf{Int}$, and then we combine these types as an intersection type $(\sigma \to \mathsf{Int}) \wedge (\tau \to \mathsf{Int})$,
3. we have

$$(\sigma \to \mathsf{Int}) \wedge (\tau \to \mathsf{Int}) \quad \leqslant \quad (\sigma \vee \tau) \to \mathsf{Int},$$

   so the standard rule for function application gives that E has type Int.

Details of how to do the first two of these steps follow. Good choices of σ and τ are:

$$\begin{aligned} \sigma &= \mu\alpha.((\mathsf{Int} \to \mathsf{Int}) \wedge (\alpha \to \alpha)) \\ \tau &= \mu\beta.(\beta \to (\mathsf{Int} \to \mathsf{Int})). \end{aligned}$$

These two types are not found by clever guessing; rather we first performed an Agesen-style analysis, see section 4, and then we mapped the obtained flows to types, see section 5. The resulting types for $\lambda x.x$ and $\lambda y.\lambda z.z$ are σ and τ, respectively. Notice that these recursive types satisfy the equations $\sigma = (\mathsf{Int} \to \mathsf{Int}) \wedge (\sigma \to \sigma)$ and $\tau = \tau \to (\mathsf{Int} \to \mathsf{Int})$. By type checking $\lambda x.x$ twice we see that it has type σ, and by analyzing $\lambda y.\lambda z.z$ just once, we see that it has type τ. Now let us type check $\lambda f.\mathsf{succ}\ ((ff)\mathsf{0})$ twice, and let us use subtyping to do that. First, we want to derive $f : \sigma \vdash ff : \mathsf{Int} \to \mathsf{Int}$. The type derivation is:

$$\dfrac{\dfrac{\dfrac{f : \sigma \vdash f : \sigma}{f : \sigma \vdash f : \sigma \to \sigma} \qquad f : \sigma \vdash f : \sigma}{f : \sigma \vdash ff : \sigma}}{f : \sigma \vdash ff : \mathsf{Int} \to \mathsf{Int}}$$

where we have used that

$$\begin{aligned} \sigma &= (\mathsf{Int} \to \mathsf{Int}) \wedge (\sigma \to \sigma) \quad \leqslant \quad \sigma \to \sigma \\ \sigma &= (\mathsf{Int} \to \mathsf{Int}) \wedge (\sigma \to \sigma) \quad \leqslant \quad \mathsf{Int} \to \mathsf{Int}. \end{aligned}$$

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

Second we want to derive $f : \tau \vdash ff : \mathsf{Int} \to \mathsf{Int}$. The type derivation is:

$$\frac{f : \tau \vdash f : \tau \to (\mathsf{Int} \to \mathsf{Int}) \qquad f : \tau \vdash f : \tau}{f : \tau \vdash ff : \mathsf{Int} \to \mathsf{Int}}$$

where we have used that $\tau = \tau \to (\mathsf{Int} \to \mathsf{Int})$. It is now straightforward to derive $\emptyset \vdash \lambda f.\mathsf{succ}\ ((ff)\mathsf{0}) : (\sigma \to \mathsf{Int}) \wedge (\tau \to \mathsf{Int})$. Conclusion: E is typable.

Alternative choices of σ and τ are:

$$\begin{aligned} \sigma &= (\mathsf{Int} \to \mathsf{Int}) \wedge ((\mathsf{Int} \to \mathsf{Int}) \to (\mathsf{Int} \to \mathsf{Int})) \\ \tau &= (\mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}) \wedge ((\mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int} \to \mathsf{Int}). \end{aligned}$$

Notice that these types are not recursive. It remains open if there is a way of mapping the flow information from the Agesen-style analysis to these types.

2 The example language

Our example language is a set of labeled λ-terms, defined by the following grammar which is in the style of Nielson and Nielson (1997):

$$\begin{array}{lcll} e & \in & \mathit{Exp} & \text{(labeled terms)} \\ e & ::= & t^l & \\ t & \in & \mathit{Term} & \text{(unlabeled expressions)} \\ t & ::= & x \mid \lambda x.e \mid e_1 e_2 \mid c \mid \mathsf{succ}\ e \mid \mathsf{if0}\ e_1\ e_2\ e_3 & \\ l & \in & \mathit{Lab} & \text{(infinite set of labels)} \\ x & \in & \mathit{Var} & \text{(infinite set of variables)} \\ c & \in & \mathit{IntegerConstant} & \end{array}$$

A *program* is a closed expression. Let *Abs* denote the set of elements of *Exp* of the form $(\lambda x.e)^l$, and let *LabelledIntegerConstant* denote the set of elements of *Exp* of the form c^l. A *value* is an element of $\mathit{Abs} \cup \mathit{LabelledIntegerConstant}$. We use v to range over values. We use $\lceil c \rceil$ to denote the integer represented by an integer constant c. A small-step call-by-value operational semantics for the language is given by the reflexive, transitive closure of the relation $\to_V$:

$$\to_V \subseteq \mathit{Exp} \times \mathit{Exp}$$

$$((\lambda x.e)^{l_1}\ v)^l \to_V e[x := v] \tag{1}$$

$$\frac{e_1 \to_V e_3}{(e_1 e_2)^l \to_V (e_3 e_2)^l} \tag{2}$$

$$\frac{e_2 \to_V e_4}{(v\ e_2)^l \to_V (v\ e_4)^l} \tag{3}$$

$$(\mathsf{succ}\ c_1^{l_1})^l \to_V c_2^l \quad (\lceil c_2 \rceil = \lceil c_1 \rceil + 1) \tag{4}$$

$$\frac{e_1 \to_V e_2}{(\mathsf{succ}\ e_1)^l \to_V (\mathsf{succ}\ e_2)^l} \tag{5}$$

$$(\mathsf{if0}\ c^{l_1}\ e_2\ e_3)^l \to_V e_2 \quad (\lceil c \rceil = 0) \tag{6}$$

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

$$(\mathsf{if0}\ c^{l_1}\ e_2\ e_3)^l \to_V e_3 \quad (\lceil c \rceil \neq 0) \tag{7}$$

$$\frac{e_1 \to_V e_4}{(\mathsf{if0}\ e_1\ e_2\ e_3)^l \to_V (\mathsf{if0}\ e_4\ e_2\ e_3)^l} \tag{8}$$

The notation $e[x := e']$ denotes e with every free occurrence of x substituted by e':

$$\begin{aligned}
x^l[x := e'] &\equiv e' \\
y^l[x := e'] &\equiv y^l \quad (x \not\equiv y) \\
(\lambda x.e_1)^l[x := e'] &\equiv (\lambda x.e_1)^l \\
(\lambda y.e_1)^l[x := e'] &\equiv (\lambda z.((e_1[y \mapsto z])[x := e']))^l \quad (x \not\equiv y \text{ and } z \text{ is fresh}) \\
(e_1\ e_2)^l[x := e'] &\equiv ((e_1[x := e'])\ (e_2[x := e']))^l \\
c^l[x := e'] &\equiv c^l \\
(\mathsf{succ}\ e_1)^l[x := e'] &\equiv (\mathsf{succ}\ (e_1[x := e']))^l \\
(\mathsf{if0}\ e_1\ e_2\ e_3)^l[x := e'] &\equiv (\mathsf{if0}\ (e_1[x := e'])\ (e_2[x := e'])\ (e_3[x := e']))^l
\end{aligned}$$

where, for a fresh variable z, the notation $e[y \mapsto z]$ denotes e with every free occurrence of y renamed to z:

$$\begin{aligned}
y^l[y \mapsto z] &\equiv z^l \\
x^l[y \mapsto z] &\equiv x^l \quad (x \not\equiv y) \\
(\lambda y.e_1)^l[y \mapsto z] &\equiv (\lambda y.e_1)^l \\
(\lambda x.e_1)^l[y \mapsto z] &\equiv (\lambda x.(e_1[y \mapsto z]))^l \quad (x \not\equiv y) \\
(e_1\ e_2)^l[y \mapsto z] &\equiv ((e_1[y \mapsto z])\ (e_2[y \mapsto z]))^l \\
c^l[y \mapsto z] &\equiv c^l \\
(\mathsf{succ}\ e_1)^l[y \mapsto z] &\equiv (\mathsf{succ}\ (e_1[y \mapsto z]))^l \\
(\mathsf{if0}\ e_1\ e_2\ e_3)^l[y \mapsto z] &\equiv (\mathsf{if0}\ (e_1[y \mapsto z])\ (e_2[y \mapsto z])\ (e_3[y \mapsto z]))^l.
\end{aligned}$$

It is convenient to define substitution and renaming separately because substitution changes both the name and label of x^l (the rule $x^l[x := e'] \equiv e'$), whereas renaming preserves the label (the rule $y^l[y \mapsto z] \equiv z^l$).

Lemma 2.1
If e is closed, and $e \to_V e'$, then e' is closed.

Proof
This is a well-known lemma for λ-calculi where the λ-terms are unlabeled, see (Barendregt, 1981). The proof is by induction on the structure of the derivation of $e \to_V e'$. We omit the details. □

An expression e is *stuck* if and only if it is not a value and there is no expression e' such that $e \to_V e'$. A program *goes wrong* if and only if it evaluates to a stuck expression. Examples of stuck expressions include $(c^{l_1} v)^l$, $(\mathsf{succ}\ (\lambda x.e)^{l_1})^l$, and $(\mathsf{if0}\ (\lambda x.e)^{l_1}\ e_2\ e_3)^l$. Intuitively, these expressions are stuck because c is not a function, succ cannot be applied to functions, and $\lambda x.e$ is not an integer.

Notice that variables are not values, because our programs are closed expressions, and we only allow β-reduction outside the bodies of λ-abstractions. If β-reduction

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

is allowed in the body of λ-abstractions, then the operand-part of a β-redex can be open, even though the whole program is closed. For example, in $(\lambda x.((\lambda y.y^1)^2 x^3)^4)^5$, the operand-part of $((\lambda y.y)^2 x^3)^4$ is a variable. In such a setting, it would be natural to let variables be values.

We chose to work with a call-by-value operational semantics because we can prove that types are preserved during evaluation. It has been observed by Barbanera *et al.* (1995), and also by Wells *et al.* (1997) that in some type systems with intersection and union types, types are not preserved by more general notions of reduction.

3 The type system

The goal of this section is to define and prove the correctness of the type system $\mathscr{T}_{\leqslant_1}$. We will do that in two steps. First, we define a type system $\mathscr{T}_{\leqslant}$ which is parameterized by a type ordering $\leqslant$. We prove that if $\leqslant$ is *acceptable*, then a program typable in $\mathscr{T}_{\leqslant}$ cannot go wrong (Corollary 3.9). Secondly, we define the type ordering $\leqslant_1$ and prove that it is acceptable (Theorem 3.12).

3.1 Terms

Following Kozen *et al.* (1995), we give a general definition of (possibly infinite) terms over an arbitrary finite ranked alphabet Σ. Such terms are essentially labeled trees, which we represent as partial functions labeling strings over ω (the natural numbers) with elements of Σ.

Let Σ_n denote the set of elements of Σ of arity n. Let ω denote the set of positive natural numbers and let ω^* denote the set of finite-length strings over ω.

A *term* over Σ is a partial function

$$t : \omega^* \quad \rightarrow \quad \Sigma$$

with domain $\mathscr{D}(t)$ satisfying the following properties:

- $\mathscr{D}(t)$ is nonempty and prefix-closed;
- if $t(\alpha) \in \Sigma_n$, then $\{i \mid \alpha i \in \mathscr{D}(t)\} = \{1, 2, \ldots, n\}$.

Let t be a term and $\alpha \in \omega^*$. Define the partial function $t\downarrow\alpha : \omega^* \rightarrow \Sigma$ by

$$t\downarrow\alpha(\beta) \quad = \quad t(\alpha\beta).$$

If $t \downarrow \alpha$ has nonempty domain, then it is a term, and is called the *subterm of* t *at position* α.

A term t is said to be *regular* if it has only finitely many distinct subterms; *i.e.*, if $\{t\downarrow\alpha \mid \alpha \in \omega^*\}$ is a finite set. Courcelle (1983) showed that t is regular if and only if t can be described by a finite set of equations involving the μ operator.

3.2 Types

In this subsection we define a set of types, where each type is of one of the forms:

$$\bigvee_{i \in I} \bigwedge_{k \in K} (\sigma_{ik} \rightarrow \sigma'_{ik})$$

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

$$(\bigvee_{i\in I} \bigwedge_{k\in K} (\sigma_{ik} \to \sigma'_{ik})) \vee \mathsf{Int}.$$

In the case of $I = \emptyset$, the first form can be simplified to $\bot$, and the second form can be simplified to Int. We will also define a notion of type equality for such types. We use the style of definition used by Palsberg and Zhao (2000).

A type is a regular term over the ranked alphabet

$$\Sigma = \{\mathsf{Int}, \bot, \to\} \ \cup \ \{\wedge^n, n \geqslant 2\} \ \cup \ \{\vee^n, n \geqslant 2\},$$

where Int, $\bot$ are nullary, $\to$ is binary, and $\vee^n, \wedge^n$ are of n-ary.

We impose the restrictions that given a type σ and a path α, if $\sigma(\alpha) = \vee^n$, then $\sigma(\alpha i) \in \{\mathsf{Int}, \bot, \to\} \ \cup \ \{\wedge^n, n \geqslant 2\}$, for all $i \in \{1..n\}$, and if $\sigma(\alpha) = \wedge^n$, then $\sigma(\alpha i) = \to$, for all $i \in \{1..n\}$.

Given a type σ, if $\sigma(\epsilon) = \to$, $\sigma(1) = \sigma_1$, and $\sigma(2) = \sigma_2$, then we write the type as $\sigma_1 \to \sigma_2$. If $\sigma(\epsilon) = \wedge^n$ and $\sigma(i) = \sigma_i \ \forall i \in \{1, 2, \ldots, n\}$, then we write the type σ as $\wedge_{i=1}^n \sigma_i$. If $\sigma(\epsilon) = \vee^n$ and $\sigma(i) = \sigma_i \ \forall i \in \{1, 2, \ldots, n\}$, then we write the type σ as $\vee_{i=1}^n \sigma_i$. If $\sigma(\epsilon) = \bot$, then we write the type as $\bot$. If $\sigma(\epsilon) = \mathsf{Int}$, then we write the type as Int.

The set of types is denoted *Type*. We use δ, σ, τ to range over types. We define the set of intersection types to be a subset of *Type*:

$$\mathit{IntersectionType} = \{ \ \sigma \in \mathit{Type} \ \mid \ \sigma(\epsilon) = \wedge^n, \text{ for some } n, \text{ or } \sigma(\epsilon) = \to \ \}.$$

We use T to range over intersection types. We will use Int as the type of integer constants, and we will use intersection types as the types of λ-abstractions. We use u to range over $\{\mathsf{Int}\} \cup \mathit{IntersectionType}$.

Intuitively, our restrictions mean that neither union types nor intersection types can be immediately nested, that is, one cannot form a union type one of whose immediate components is again a union type, and similarly for intersection types. We impose this restriction for two reasons:

1. it effectively rules out infinite intersection and union types, and
2. it ensures that types are in a 'normal form' with respect to associativity and commutativity, i.e. the issues of associativity and commutativity are reduced to a matter of the order of the components in a $\wedge_{i=1}^n \sigma_i$ type and a $\vee_{i=1}^n \sigma_i$ type.

We now define type equality. A relation R is called a *bisimulation* if it satisfies the following conditions:

1. If $(\vee_{i=1}^n \sigma_i, \vee_{j=1}^m \tau_j) \in R$, then
   - for all $i \in \{1..n\}$, where $\sigma_i(\epsilon) \neq \bot$: there exists $j \in \{1..m\} : (\sigma_i, \tau_j) \in R$, and
   - for all $j \in \{1..m\}$, where $\tau_j(\epsilon) \neq \bot$, there exists $i \in \{1..n\} : (\sigma_i, \tau_j) \in R$.
2. If $\tau(\epsilon) \in \{\mathsf{Int}, \bot, \to\} \cup \{\wedge^m, m \geqslant 2\}$, and $(\vee_{i=1}^n \sigma_i, \tau) \in R$, then,
   - for all $i \in \{1..n\}$, where $\sigma_i(\epsilon) \neq \bot$: $(\sigma_i, \tau) \in R$, and
   - if $\tau(\epsilon) \neq \bot$, then there exists $i \in \{1..n\} : (\sigma_i, \tau) \in R$.
3. If $\tau(\epsilon) \in \{\mathsf{Int}, \bot, \to\} \cup \{\wedge^m, m \geqslant 2\}$, and $(\tau, \vee_{i=1}^n \sigma_i) \in R$, then,
   - for all $i \in \{1..n\}$, where $\sigma_i(\epsilon) \neq \bot$: $(\tau, \sigma_i) \in R$, and

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

- if $\tau(\epsilon) \neq \bot$, then there exists $i \in \{1..n\} : (\tau, \sigma_i) \in R$.

4. If $(\wedge_{i=1}^{n}\sigma_i, \wedge_{j=1}^{n}\tau_j) \in R$, then there exists a bijection $t : \{1..n\} \to \{1..n\}$ such that for all $i \in \{1..n\} : (\sigma_i, \tau_{t(i)}) \in R$.
5. If $(\sigma_1 \to \sigma_2, \tau_1 \to \tau_2) \in R$, then $(\sigma_1, \tau_1) \in R$ and $(\sigma_2, \tau_2) \in R$.
6. If $(\sigma, \tau) \in R$, then either

$$\begin{aligned}
&\sigma = \tau = \bot \\
&\sigma = \tau = \mathsf{Int} \\
&\sigma(\epsilon) = \tau(\epsilon) = \to \\
&\sigma(\epsilon) = \tau(\epsilon) = \wedge^n \\
&\sigma(\epsilon) = \vee^n, \text{ or} \\
&\tau(\epsilon) = \vee^n.
\end{aligned}$$

Bisimulation are closed under union, therefore, there exists a largest bisimulation

$$\mathscr{E} = \bigcup\{\, R \mid R \text{ is a bisimulation}\}.$$

The set $\mathscr{E}$ is our notion of type equality. We may apply the principle of *co-induction* to prove that two types are related in $\mathscr{E}$, that is, to show $(\sigma, \tau) \in \mathscr{E}$, it is sufficient to find a bisimulation R such that $(\sigma, \tau) \in R$.

Theorem 3.1
The following assertions are true:

- $\mathscr{E}$ is a congruence relation,
- if $\wedge_{i=1}^{n}\sigma_i$ is a type and $t : \{1..n\} \to \{1..n\}$ is a bijection, then $(\wedge_{i=1}^{n}\sigma_i, \wedge_{i=1}^{n}\sigma_{t(i)}) \in \mathscr{E}$, and
- if $\vee_{i=1}^{n}\sigma_i$ is a type and $t : \{1..n\} \to \{1..n\}$ is a bijection, then $(\vee_{i=1}^{n}\sigma_i, \vee_{i=1}^{n}\sigma_{t(i)}) \in \mathscr{E}$.

Proof
By co-induction, we omit the details. □

Type equality as defined by $\mathscr{E}$ can be decided in polynomial time. The case without union types is covered by the algorithm of Palsberg and Zhao (2000), and it is straightforward to extend their algorithm to handle our notion of union types, we omit the details.

We use I, J to range over finite and possibly empty index sets. We use K to range over finite and nonempty index sets. From Theorem 3.1 we have that for type equality the ordering of the components is not important when considering intersection and union types. So, we will use the notation

$$\bigwedge_{k\in K}\sigma_k \qquad \bigvee_{i\in I}\sigma_i$$

to denote intersection types and union types where the orderings of the components are left unspecified. By convention, if K is a singleton set, say $K = \{k_0\}$, then $\bigwedge_{k\in K}\sigma_k$ denotes σ_{k_0}. Similarly, if I is a singleton set, say $I = \{i_0\}$, then $\bigvee_{i\in I}\sigma_i$ denotes σ_{i_0}. Moreover, if $I = \emptyset$, then $\bigvee_{i\in I}\sigma_i$ denotes $\bot$.

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

We will use the notation

$$\bigwedge_{k=1}^{n}(\sigma_k \to \sigma'_k) \quad \equiv \quad (\sigma_1 \to \sigma'_1) \wedge \ldots \wedge (\sigma_n \to \sigma'_n)$$

$$\bigvee_{i=1}^{n}\sigma_i \quad \equiv \quad \sigma_1 \vee \ldots \vee \sigma_n.$$

For union types, we will even go a bit further and allow a notation where the binary $\vee$ is applied to two types that may be union types. This should be seen as a shorthand for a 'flattened' type that satisfies the restriction that union types cannot be nested. It is straightforward to show, by co-induction, that

$$\sigma \vee \bot \quad = \quad \bot \vee \sigma \; = \; \sigma \vee \sigma \; = \; \sigma.$$

3.3 Type orderings

Pierce studied type orderings on intersection and union types (Pierce, 1991). Here we begin by stating conditions on type orderings which will be sufficient to prove type soundness. Later we will give examples of type orderings which satisfy the conditions.

Definition 3.2
(Acceptable Type Orderings) We say that an ordering $\leqslant$ on types is *acceptable* if and only if $\leqslant$ satisfies the five conditions:

1. $\leqslant$ is reflexive,
2. $\leqslant$ is transitive,
3. if $\bigwedge_{k\in K}(\sigma_k \to \sigma'_k) \leqslant (\tau_1 \to \tau_2)$, and $u \leqslant \tau_1$, then there exists $k_0 \in K$ such that $u \leqslant \sigma_{k_0}$ and $\sigma'_{k_0} \leqslant \tau_2$,
4. $\bigwedge_{k\in K}(\sigma_k \to \tau_k) \not\leqslant \mathsf{Int}$, and
5. $\mathsf{Int} \not\leqslant \sigma \to \tau$.

For example, the identity relation on types is an acceptable type ordering. If $\leqslant$ is an acceptable type ordering, and $\sigma \leqslant \tau$ then we say that 'σ is a subtype of τ'.

3.4 Type rules

If $\mathscr{F}$ is a partial function from $\mathscr{D}_1$ to a set $\mathscr{D}_2$, and $d \in \mathscr{D}_1$, then $\mathscr{F}[x : d]$ denotes a partial function which maps x to d, and maps y, where $y \neq x$, to $\mathscr{F}(y)$. If $\mathscr{F}$ is a partial function, then $dom(\mathscr{F})$ denotes the set of points on which $\mathscr{F}$ is defined. We use $\emptyset$ to denote the partial function for which $dom(\emptyset) = \emptyset$. We use $\hookrightarrow$ to denote the constructor of spaces of partial functions with finite domain. Define

$$\begin{array}{lcll} A & \in & \mathit{TypeEnv} & = & \mathit{Var} \hookrightarrow \mathit{Type} \\ & & \mathit{TypeJudgment} & = & \mathit{TypeEnv} \times \mathit{Exp} \times \mathit{Type}. \end{array}$$

A type environment is a partial function from variables to types. A type judgment is a triple which will be written $A \vdash e : \tau$. Intuitively, such a type judgment indicates that in the type environment A, the expression e has type τ.

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

We now define a type system $\mathscr{T}_{\leqslant}$ which is parameterized by a type ordering $\leqslant$. Given a type ordering $\leqslant$, we will inductively define a set of valid type judgments. We write $\mathscr{T}_{\leqslant} \triangleright A \vdash e : \tau$ if and only if the judgment $A \vdash e : \tau$ follows by Rules (9)–(15), where all applications of Rule (15) use the ordering $\leqslant$.

$$A[x : \tau] \vdash x^l : \tau \tag{9}$$

$$\frac{\forall k \in K : A[x : \sigma_k] \vdash e : \tau_k}{A \vdash (\lambda x.e)^l : \bigwedge_{k \in K}(\sigma_k \to \tau_k)} \tag{10}$$

$$\frac{A \vdash e_1 : \sigma \to \tau \quad A \vdash e_2 : \sigma}{A \vdash (e_1 e_2)^l : \tau} \tag{11}$$

$$A \vdash c^l : \mathsf{Int} \tag{12}$$

$$\frac{A \vdash e : \mathsf{Int}}{A \vdash (\mathsf{succ}\ e)^l : \mathsf{Int}} \tag{13}$$

$$\frac{A \vdash e_1 : \mathsf{Int} \quad A \vdash e_2 : \tau \quad A \vdash e_3 : \tau}{A \vdash (\mathsf{if0}\ e_1\ e_2\ e_3)^l : \tau} \tag{14}$$

$$\frac{A \vdash e : \sigma}{A \vdash e : \tau} \quad (\sigma \leqslant \tau) \tag{15}$$

Rule (10) is unusual because it assigns a λ-abstraction an intersection of function types rather than a single function type.

We say that e is *typable* in $\mathscr{T}_{\leqslant}$ if and only if there exist A, τ such that $\mathscr{T}_{\leqslant} \triangleright A \vdash e : \tau$.

3.5 Correctness

We will now prove that for an acceptable type ordering $\leqslant$, a program typable in $\mathscr{T}_{\leqslant}$ cannot go wrong. We use the proof technique of Nielson (1989) and others that was popularized by Wright and Felleisen (1994).

Lemma 3.3
(Strengthening) If $\mathscr{T}_{\leqslant} \triangleright A[x : \sigma] \vdash e : \tau$, and x does not occur free in e, then $\mathscr{T}_{\leqslant} \triangleright A \vdash e : \tau$.

Proof
The proof is by a straightforward induction on the structure of the derivation of $A[x : \sigma] \vdash e : \tau$. We omit the details. □

Lemma 3.4
(Renaming) If $\mathscr{T}_{\leqslant} \triangleright A[x : \sigma] \vdash e : \tau$, and z does not occur free in e, then $\mathscr{T}_{\leqslant} \triangleright A[z : \sigma] \vdash e[x \mapsto z] : \tau$ in such a way that the two derivations are of the same height.

Proof
The proof is by a straightforward induction on the structure of the derivation of $A[x : \sigma] \vdash e : \tau$. We omit the details. □

Lemma 3.5
(Substitution) If $\mathscr{T}_{\leqslant} \triangleright A[x : \sigma] \vdash e : \tau$ and $\mathscr{T}_{\leqslant} \triangleright A \vdash e' : \sigma$, then $\mathscr{T}_{\leqslant} \triangleright A \vdash e[x := e'] : \tau$.

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

Proof
We proceed by induction on the height of the derivation of $A[x : \sigma] \vdash e : \tau$. There are now seven subcases depending on which one of Rules (9)–(15) was the last one used in the derivation of $A[x : \sigma] \vdash e : \tau$.

- Rule (9). We have $e \equiv y^l$. There are two subcases.
  - $x \equiv y$. We have $y^l[x := e'] \equiv e'$. The whole derivation of $A[x : \sigma] \vdash e : \tau$ is of the form
    $$A[y : \tau] \vdash y^l : \tau$$
    so $\sigma = \tau$, and therefore the desired conclusion is identical to the second hypothesis.
  - $x \not\equiv y$. We have $y^l[x := e'] \equiv y^l$. The whole derivation of $A[x : \sigma] \vdash e : \tau$ is of the form
    $$A'[y : \tau][x : \sigma] \vdash y^l : \tau,$$
    where $A = A'[y : \tau]$, and from Rule (9) we derive $A'[y : \tau] \vdash y^l : \tau$.
- Rule (10). We have $e \equiv (\lambda y.e_1)^l$. There are two subcases.
  - $x \equiv y$. We have $(\lambda y.e_1)^l[x := e'] \equiv (\lambda y.e_1)^l$. Since x does not occur free in $(\lambda y.e_1)^l$, we can from the derivation of $A[x : \sigma] \vdash (\lambda y.e_1)^l : \tau$ and Lemma 3.3 produce a derivation of $A \vdash (\lambda y.e_1)^l : \tau$.
  - $x \not\equiv y$. We have $(\lambda y.e_1)^l[x := e'] \equiv (\lambda z.((e_1[y \mapsto z])[x := e']))^l$, where z is fresh. The last step in the derivation of $A[x : \sigma] \vdash e : \tau$ is of the form
    $$\frac{\forall k \in K : A[x : \sigma][y : \sigma_k] \vdash e_1 : \tau_k}{A[x : \sigma] \vdash (\lambda y.e_1)^l : \bigwedge_{k \in K}(\sigma_k \to \tau_k)}$$
    For all $k \in K$, from the derivation of $A[x : \sigma][y : \sigma_k] \vdash e_1 : \tau_k$, and Lemma 3.4, we can produce a derivation of $A[x : \sigma][z : \sigma_k] \vdash e_1[y \mapsto z] : \tau_k$ of the same height. From the induction hypothesis we have that we can derive $A[z : \sigma_k] \vdash e_1[y \mapsto z][x := e'] : \tau_k$, so from Rule (10) we can derive $A \vdash (\lambda z.((e_1[y \mapsto z])[x := e']))^l : \bigwedge_{k \in K}(\sigma_k \to \tau_k)$.
- Rule (11). We have $e \equiv (e_1\ e_2)^l$, and $(e_1\ e_2)^l[x := e'] \equiv ((e_1[x := e'])\ (e_2[x := e']))^l$. The last step in the derivation of $A[x : \sigma] \vdash e : \tau$ is of the form
  $$\frac{A[x : \sigma] \vdash e_1 : \tau_2 \to \tau \quad A[x : \sigma] \vdash e_2 : \tau_2}{A[x : \sigma] \vdash (e_1\ e_2)^l : \tau}$$
  From the induction hypothesis we have that we can derive $A \vdash e_1[x := e'] : \tau_2 \to \tau$ and $A \vdash e_2[x := e'] : \tau_2$, and from Rule (11) we can derive $A \vdash ((e_1[x := e'])\ (e_2[x := e']))^l : \tau$.
- Rule (12). We have $e \equiv c^l$, and $c^l[x := e'] \equiv c^l$. The last step in the derivation of $A[x : \sigma] \vdash e : \tau$ is of the form
  $$A[x : \sigma] \vdash c^l : \mathsf{Int}$$
  and from Rule (12) we can derive $A \vdash c^l : \mathsf{Int}$.
- Rule (13), Rule (14), Rule (15). Each of these cases is similar to the case of Rule (11). We omit the details.

□

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

We will say that a type derivation is in *canonical form* if and only if each application of Rules (9)–(14) is followed by exactly one application of Rule (15), possibly except the last application of one of Rules (9)–(14). Notice that if a type derivation is in canonical form, then all its subtrees are also in canonical form. For a reflexive and transitive type ordering $\leqslant$, if we have a type derivation of $A \vdash e : \tau$, then we can transform it into a canonical-form derivation of $A \vdash e : \tau$.

Theorem 3.6
(Type Preservation) For an acceptable type ordering $\leqslant$, if $\mathscr{T}_{\leqslant} \triangleright A \vdash e : \tau$ and $e \rightarrow_V e'$, then $\mathscr{T}_{\leqslant} \triangleright A \vdash e' : \tau$.

Proof
It is sufficient to show the result for all canonical-form derivations of $A \vdash e : \tau$. We proceed by induction on the structure of the derivation of $A \vdash e : \tau$. There are now seven subcases depending on which one of Rules (9)–(15) was the last one used in the derivation of $A \vdash e : \tau$.

- Rule (9). We have $e \equiv x^l$, so $e \rightarrow_V e'$ is not possible.
- Rule (10). We have $e \equiv (\lambda x.e_1)^l$, so $e \rightarrow_V e'$ is not possible.
- Rule (11). We have $e \equiv (e_1 e_2)^l$. There are now three subcases depending on which one of Rules (1)–(3) was the last one used in the derivation of $e \rightarrow_V e'$.
  - — Rule (1). We have $e \equiv ((\lambda x.e_1)^{l_1} v)^l$ and $e' \equiv e_1[x := v]$. The last part of the derivation of $A \vdash e : \tau$ is of the form
$$\dfrac{\dfrac{\dfrac{\forall k \in K : A[x : \sigma_k] \vdash e_1 : \tau_k}{A \vdash (\lambda x.e_1)^{l_1} : \bigwedge_{k \in K}(\sigma_k \rightarrow \tau_k)} \quad \dfrac{A \vdash v : u}{A \vdash v : \sigma}}{A \vdash (\lambda x.e_1)^{l_1} : \sigma \rightarrow \tau}}{A \vdash ((\lambda x.e_1)^{l_1} v)^l : \tau}$$
    where $\bigwedge_{k \in K}(\sigma_k \rightarrow \tau_k) \leqslant \sigma \rightarrow \tau$ and $u \leqslant \sigma$. From Definition 3.2 (condition 3) we have that there exists $k_0 \in K$ such that $u \leqslant \sigma_{k_0}$ and $\tau_{k_0} \leqslant \tau$. From $A \vdash v : u$ and $u \leqslant \sigma_{k_0}$ and Rule (15) we derive $A \vdash v : \sigma_{k_0}$. From Lemma 3.5, $A[x : \sigma_{k_0}] \vdash e_1 : \tau_{k_0}$, and $A \vdash v : \sigma_{k_0}$ we derive $A \vdash e_1[x := v] : \tau_{k_0}$. From Rule (15) and $\tau_{k_0} \leqslant \tau$, we can finally derive $A \vdash e_1[x := v] : \tau$.
  - — Rules (2)–(3). In each case a derivation of $A \vdash e' : \tau$ is provided by the induction hypothesis and Rule (11).
- Rule (12). We have $e \equiv c^l$, so $e \rightarrow_V e'$ is not possible.
- Rule (13). We have $e \equiv (\mathsf{succ}\ e_1)^l$. There are now two subcases depending on which one of Rules (4)–(5) was the last one used in the derivation of $e \rightarrow_V e'$.
  - — Rule (4). We have $e \equiv (\mathsf{succ}\ c_1^{l_1})^l$ and $e' \equiv c_2^l$, where $\lceil c_2 \rceil = \lceil c_1 \rceil + 1$. The last type judgment in the derivation of $A \vdash e : \tau$ is of the form $A \vdash (\mathsf{succ}\ c_1^{l_1})^l : \mathsf{Int}$, and from Rule (12) we derive $A \vdash c_2^l : \mathsf{Int}$.
  - — Rule (5). We have $e \equiv (\mathsf{succ}\ e_1)^l$ and $e' \equiv (\mathsf{succ}\ e_2)^l$, and $e_1 \rightarrow_V e_2$. The last part of the derivation of $A \vdash e : \tau$ is of the form
$$\dfrac{A \vdash e_1 : \mathsf{Int}}{A \vdash (\mathsf{succ}\ e_1)^l : \mathsf{Int}}$$
    The induction hypothesis provides a derivation of $A \vdash e_2 : \mathsf{Int}$, and from Rule (13) we derive $A \vdash (\mathsf{succ}\ e_2)^l : \mathsf{Int}$.

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

- Rule (14). We have $e \equiv (\mathsf{if0}\ e_1\ e_2\ e_3)^l$. There are now three subcases depending on which one of Rules (6)–(8) was the last one used in the derivation of $e \rightarrow_V e'$. In each case $A \vdash e' : \tau$ is easily derived using the induction hypothesis.
- Rule (15). The last part of the derivation of $A \vdash e : \tau$ is of the form

$$\frac{A \vdash e : \sigma}{A \vdash e : \tau} \quad (\sigma \leqslant \tau)$$

and from $A \vdash e : \sigma$, the induction hypothesis provides a derivation of $A \vdash e' : \sigma$. From this and Rule (15) we derive $A \vdash e' : \tau$.

□

Lemma 3.7
For an acceptable type ordering $\leqslant$, if $\mathscr{T}_{\leqslant} \triangleright A \vdash v : \mathsf{Int}$, then v is of the form c^l; and if $\mathscr{T}_{\leqslant} \triangleright A \vdash v : \sigma \rightarrow \tau$, then v is of the form $(\lambda x.e)^l$.

Proof
Suppose first that we have a canonical-form derivation of $A \vdash v : \mathsf{Int}$. If $v \equiv (\lambda x.e)^l$, then the last part of the derivation of $A \vdash v : \mathsf{Int}$ is of the form

$$\frac{A \vdash (\lambda x.e)^l : \bigwedge_{k \in K} (\sigma_k \rightarrow \tau_k)}{A \vdash (\lambda x.e)^l : \mathsf{Int}}$$

where $\bigwedge_{k \in K} (\sigma_k \rightarrow \tau_k) \leqslant \mathsf{Int}$. ¿From Definition 3.2 (condition 4) we have that this is impossible, so the assumption that $v \equiv (\lambda x.e)^l$ is wrong, and hence v is must be of the form c^l.

Suppose then that we have a canonical-form derivation of $A \vdash v : \sigma \rightarrow \tau$. If $v \equiv c^l$, then the last part of the derivation of $A \vdash v : \sigma \rightarrow \tau$ is of the form

$$\frac{A \vdash c^l : \mathsf{Int}}{A \vdash c^l : \sigma \rightarrow \tau}$$

where $\mathsf{Int} \leqslant \sigma \rightarrow \tau$. From Definition 3.2 (condition 5) we have that this is impossible, so the assumption that $v \equiv c^l$ is wrong, and hence v is must be of the form $(\lambda x.e)^l$.
□

The following lemma states that a typable program is not stuck.

Lemma 3.8
(Progress) For an acceptable type ordering $\leqslant$, if e is a closed expression, and $\mathscr{T}_{\leqslant} \triangleright A \vdash e : \tau$, then either e is a value, or there exists e' such that $e \rightarrow_V e'$.

Proof
We proceed by induction on the structure of the derivation of $A \vdash e : \tau$. There are now seven subcases depending on which one of Rules (9)–(15) was the last one used in the derivation of $A \vdash e : \tau$.

- Rule (9). We have $e \equiv x$, and x is not closed.
- Rule (10). We have $e \equiv (\lambda x.e_1)^l$, so e is a value.

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

- Rule (11). We have $e \equiv (e_1 e_2)^l$. We have that e is closed, so also e_1, e_2 are closed. The last step in the derivation of $A \vdash e : \tau$ is of the form

$$\frac{A \vdash e_1 : \sigma \to \tau \quad A \vdash e_2 : \sigma}{A \vdash (e_1 e_2)^l : \tau}$$

  From the induction hypothesis we have that (1) either e_1 is a value, or there exists e_1' such that $e_1 \to_V e_1'$ and (2) either e_2 is a value, or there exists e_2' such that $e_2 \to_V e_2'$. We proceed by case analysis.

  — If there exists e_1' such that $e_1 \to_V e_1'$, then $(e_1 e_2)^l \to_V (e_1' e_2)^l$ by Rule (2).
  — If e_1 is a value, and there exists e_2' such that $e_2 \to_V e_2'$, then $(e_1 e_2)^l \to_V (e_1 e_2')^l$ by Rule (3).
  — If e_1, e_2 are values, then from $A \vdash e_1 : \sigma \to \tau$ and Lemma 3.7 we have that e_1 is of the form $\lambda x.e_3$, and hence $(e_1 e_2)^l \to_V e_3[x := e_2]$ by Rule (1).

- Rule (12). We have $e \equiv c^l$, so e is a value.
- Rule (13). We have $e \equiv (\mathsf{succ}\ e_1)^l$. We have that e is closed, so also e_1 is closed. The last step in the derivation of $A \vdash e : \tau$ is of the form

$$\frac{A \vdash e_1 : \mathsf{Int}}{A \vdash (\mathsf{succ}\ e_1)^l : \mathsf{Int}}$$

  From the induction hypothesis we have that either e_1 is a value, or there exists e_1' such that $e_1 \to_V e_1'$. We proceed by case analysis.

  — If e_1 is a value, then from $A \vdash e_1 : \mathsf{Int}$ and Lemma 3.7 we have that e_1 is of the form $c_1^{l_1}$, and hence $(\mathsf{succ}\ c_1^{l_1})^l \to_V c_2^l$ where $\lceil c_2 \rceil = \lceil c_1 \rceil + 1$ by Rule (4).
  — If there exists e_1' such that $e_1 \to_V e_1'$, then $(\mathsf{succ}\ e_1)^l \to_V (\mathsf{succ}\ e_1')^l$ by Rule (5).

- Rule (14). We have $e \equiv (\mathsf{if0}\ e_1\ e_2\ e_3)^l$. We have that e is closed, so also e_1, e_2, e_3 are closed. The last step in the derivation of $A \vdash e : \tau$ is of the form

$$\frac{A \vdash e_1 : \mathsf{Int} \quad A \vdash e_2 : \tau \quad A \vdash e_3 : \tau}{A \vdash (\mathsf{if0}\ e_1\ e_2\ e_3)^l : \tau}$$

  From the induction hypothesis we have that either e_1 is a value, or there exists e_1' such that $e_1 \to_V e_1'$. We proceed by case analysis.

  — If e_1 is a value, then from $A \vdash e_1 : \mathsf{Int}$ and Lemma 3.7 we have that e_1 is of the form c^{l_1} so either $e \to_V e_2$ by Rule (6) or $e \to_V e_3$ by Rule (7).
  — If there exists e_1' such that $e_1 \to_V e_1'$, then $(\mathsf{if0}\ e_1\ e_2\ e_3)^l \to_V (\mathsf{if0}\ e_1'\ e_2\ e_3)^l$ by Rule (8).

- Rule (15). The last part of the derivation of $A \vdash e : \tau$ is of the form

$$\frac{A \vdash e : \sigma}{A \vdash e : \tau} \quad (\sigma \leqslant \tau)$$

  and from $A \vdash e : \sigma$ and the induction hypothesis we have that either e is a value or there exists e' such that $e \to_V e'$.

□

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

Corollary 3.9
(Type Soundness) For an acceptable type ordering $\leqslant$, a program typable in $\mathscr{T}_{\leqslant}$ cannot go wrong.

Proof
Suppose we have a program e which is typable in $\mathscr{T}_{\leqslant}$, that is, e is closed and we have A, τ such that $\mathscr{T}_{\leqslant} \triangleright A \vdash e : \tau$. Suppose also that e can go wrong, that is, there exists a stuck expression e' such that $e \rightarrow_V^* e'$. From Lemma 2.1 we have that e' is closed. From Theorem 3.6 we have that $\mathscr{T}_{\leqslant} \triangleright A \vdash e' : \tau$. From Lemma 3.8 we have that e' is not stuck, a contradiction. We conclude that e' does not exist so e cannot go wrong. □

3.6 An acceptable type ordering

We will now define an acceptable type ordering $\leqslant_1$. We write $\sigma \leqslant_1 \tau$ if and only if we can derive $\sigma \leqslant_1 \tau$ using the following rules.

$$\frac{\sigma \leqslant_1 \delta \qquad \delta \leqslant_1 \tau}{\sigma \leqslant_1 \tau} \tag{16}$$

$$\sigma \leqslant_1 \sigma \vee \tau' \tag{17}$$

$$\frac{\forall i \in I : \sigma_i \leqslant_1 \tau_1 \rightarrow \tau_2}{\bigvee_{i \in I} \sigma_i \leqslant_1 \tau_1 \rightarrow \tau_2} \tag{18}$$

$$\frac{\tau_1 \leqslant_1 \sigma_1 \qquad \sigma_2 \leqslant_1 \tau_2}{\sigma_1 \rightarrow \sigma_2 \leqslant_1 \tau_1 \rightarrow \tau_2} \tag{19}$$

$$\frac{\bigwedge_{k \in K} (\sigma_k \rightarrow \sigma'_k) \leqslant_1 \tau_1 \rightarrow \tau_2}{\bigwedge_{k \in K'} (\sigma_k \rightarrow \sigma'_k) \leqslant_1 \tau_1 \rightarrow \tau_2} \quad (K \subseteq K') \tag{20}$$

$$\bigwedge_{k \in K} (\sigma_k \rightarrow \sigma'_k) \leqslant_1 \left(\bigvee_{k \in K} \sigma_k \right) \rightarrow \left(\bigvee_{k \in K} \sigma'_k \right) \tag{21}$$

Rule (16) is the rule for transitivity. Rule (17) is an introduction rule for union types, and Rule (18) is an elimination rule for union types. Rule (19) is the classical rule of subtyping for function types. Rules (20) and Rule (21) are elimination rules for intersection types. Rule (21) is closely related to the ($\vee$ elim)-rule in Wells *et al.* (1997). Recall that intersection types are introduced by type rule (10).

The relation $\leqslant_1$ is not antisymmetric. For example for

$$\begin{aligned} \sigma &= \mathsf{Int} \rightarrow \mathsf{Int} \\ \tau &= (\mathsf{Int} \rightarrow \mathsf{Int}) \rightarrow (\mathsf{Int} \rightarrow \mathsf{Int}) \end{aligned}$$

we have

$$\begin{aligned} \sigma &\leqslant_1 \sigma \vee (\sigma \wedge \tau) && \text{(from Rule 17)} \\ \sigma \vee (\sigma \wedge \tau) &\leqslant_1 \sigma && \text{(from Rule 18,20)} \end{aligned}$$

and yet $\sigma \neq \sigma \vee (\sigma \wedge \tau)$.

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

Lemma 3.10
(Characterization of Subtyping) We have $\sigma \leqslant_1 \tau$ if and only if

(i) either $\tau = \sigma \vee \tau'$,
(ii) or $\sigma = \bigvee_{i\in I} \bigwedge_{k\in K_i}(\sigma_{ik} \to \sigma'_{ik})$, and $\tau = (\tau_1 \to \tau_2) \vee \tau'$, and $\forall i \in I : \exists J_i : [\, \tau_1 \leqslant_1 \bigvee_{k\in J_i} \sigma_{ik}$, and $\bigvee_{k\in J_i} \sigma'_{ik} \leqslant_1 \tau_2$, and $J_i \subseteq K_i \,]$.

Proof
For $\Leftarrow$, there are two cases. If $\tau = \sigma \vee \tau'$, then we can derive $\sigma \leqslant_1 \tau$ using Rule (17).

Suppose $\sigma = \bigvee_{i\in I} \bigwedge_{k\in K_i}(\sigma_{ik} \to \sigma'_{ik})$, and $\tau = (\tau_1 \to \tau_2) \vee \tau'$, and $\forall i \in I : \exists J_i : [\, \tau_1 \leqslant_1 \bigvee_{k\in J_i} \sigma_{ik}$, and $\bigvee_{k\in J_i} \sigma'_{ik} \leqslant_1 \tau_2$, and $J_i \subseteq K_i \,]$. For all $i \in I$, we have

$$\bigwedge_{k\in J_i}(\sigma_{ik} \to \sigma'_{ik}) \quad \leqslant_1 \quad \left(\bigvee_{k\in J_i} \sigma_{ik}\right) \to \left(\bigvee_{k\in J_i} \sigma'_{ik}\right) \qquad \text{Rule (21)}$$

$$\left(\bigvee_{k\in J_i} \sigma_{ik}\right) \to \left(\bigvee_{k\in J_i} \sigma'_{ik}\right) \quad \leqslant_1 \quad \tau_1 \to \tau_2 \qquad \text{Rule (19)}$$

and from these two inequalities and Rule (16) we have

$$\bigwedge_{k\in J_i}(\sigma_{ik} \to \sigma'_{ik}) \quad \leqslant_1 \quad \tau_1 \to \tau_2.$$

For all $i \in I$, from $\bigwedge_{k\in J_i}(\sigma_{ik} \to \sigma'_{ik}) \leqslant_1 \tau_1 \to \tau_2$, $J_i \subseteq K$, and Rule (20) we have

$$\bigwedge_{k\in K}(\sigma_{ik} \to \sigma'_{ik}) \leqslant_1 \tau_1 \to \tau_2.$$

We can now use Rule (18) to derive

$$\bigvee_{i\in I} \bigwedge_{k\in K}(\sigma_{ik} \to \sigma'_{ik}) \leqslant_1 \tau_1 \to \tau_2,$$

and from Rule (17) we also have

$$\tau_1 \to \tau_2 \leqslant_1 (\tau_1 \to \tau_2) \vee \tau'.$$

Finally, an application of Rule (16) derives $\sigma \leqslant_1 \tau$.

For $\Rightarrow$, we proceed by induction on the structure of the derivation of $\sigma \leqslant_1 \tau$. There are six cases. If the last rule used in the derivation of $\sigma \leqslant_1 \tau$ is Rule (17), then we have $\tau = \sigma \vee \tau'$, so *(i)* is satisfied. If the last rule used in the derivation of $\sigma \leqslant_1 \tau$ is one of Rules (18–21), then it is in each case straightforward to show that *(ii)* is satisfied with $\tau' = \bot$. If the last rule used in the derivation of $\sigma \leqslant_1 \tau$ is Rule (16), then we have the situation

$$\frac{\sigma \leqslant_1 \delta \qquad \delta \leqslant_1 \tau}{\sigma \leqslant_1 \tau}$$

By applying the induction hypothesis to both $\sigma \leqslant_1 \delta$ and $\delta \leqslant_1 \tau$, we get that there are four subcases, which we will consider in turn.

First, suppose $\delta = \sigma \vee \delta'$, and $\tau = \delta \vee \tau'$. We have $\tau = \sigma \vee (\delta' \vee \tau')$, so *(i)* is satisfied.

Secondly, suppose $\delta = \sigma \vee \delta'$, and $\delta = \bigvee_{i\in I} \bigwedge_{k\in K_i}(\delta_{ik} \to \delta'_{ik})$, and $\tau = (\tau_1 \to \tau_2) \vee \tau'$, and $\forall i \in I : \exists J_i : [\, \tau_1 \leqslant_1 \bigvee_{k\in J_i} \delta_{ik}$, and $\bigvee_{k\in J_i} \delta'_{ik} \leqslant_1 \tau_2$, and $J_i \subseteq K_i \,]$. We

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

have $\sigma = \bigvee_{i\in I'} \bigwedge_{k\in K_i}(\delta_{ik} \to \delta'_{ik})$ where $I' \subseteq I$, so $\forall i \in I' : [\ \tau_1 \leqslant_1 \bigvee_{k\in J_i} \delta_{ik}$, and $\bigvee_{k\in J_i} \delta'_{ik} \leqslant_1 \tau_2$, and $J_i \subseteq K_i\ ]$, so *(ii)* is satisfied.

Thirdly, suppose $\sigma = \bigvee_{i\in I} \bigwedge_{k\in K_i}(\sigma_{ik} \to \sigma'_{ik})$, and $\delta = (\delta_1 \to \delta_2) \vee \delta'$, and $\forall i \in I : \exists J_i : [\ \delta_1 \leqslant_1 \bigvee_{k\in J_i} \sigma_{ik}$, and $\bigvee_{k\in J_i} \sigma'_{ik} \leqslant_1 \delta_2$, and $J_i \subseteq K_i\ ]$, and $\tau = \delta \vee \tau'$. We have $\tau = (\delta_1 \to \delta_2) \vee (\delta' \vee \tau')$, so *(ii)* is satisfied.

Fourthly, suppose $\sigma = \bigvee_{i\in I} \bigwedge_{k\in K_i}(\sigma_{ik} \to \sigma'_{ik})$, and $\delta = (\delta_1 \to \delta_2) \vee \delta'$, and $\forall i \in I : \exists J_i : [\ \delta_1 \leqslant_1 \bigvee_{k\in J_i} \sigma_{ik}$, and $\bigvee_{k\in J_i} \sigma'_{ik} \leqslant_1 \delta_2$, and $J_i \subseteq K_i\ ]$, and $\delta = \bigvee_{i\in I'} \bigwedge_{k\in K'_i}(\delta_{ik} \to \delta'_{ik})$, and $\tau = (\tau_1 \to \tau_2) \vee \tau'$, and $\forall i \in I' : \exists J'_i : [\ \tau_1 \leqslant_1 \bigvee_{k\in J'_i} \delta_{ik}$, and $\bigvee_{k\in J'_i} \delta'_{ik} \leqslant_1 \tau_2$, and $J'_i \subseteq K'_i\ ]$. From $\delta = (\delta_1 \to \delta_2) \vee \delta' = \bigvee_{i\in I'} \bigwedge_{k\in K'_i}(\delta_{ik} \to \delta'_{ik})$ we have that there exists $i_0 \in I'$ such that $\delta_1 \to \delta_2 = \bigwedge_{k\in K'_{i_0}}(\delta_{i_0 k} \to \delta'_{i_0 k})$. Moreover, for all $k \in K'_{i_0}$ we have $\delta_1 = \delta_{i_0 k}$ and $\delta_2 = \delta'_{i_0 k}$. We have $\forall i \in I : \exists J_i : [\ \tau_1 \leqslant_1 \bigvee_{k\in J'_{i_0}} \delta_{i_0 k} = \delta_1 \leqslant_1 \bigvee_{k\in J_i} \sigma_{ik}$, and $\bigvee_{k\in J_i} \sigma'_{ik} \leqslant_1 \delta_2 = \bigvee_{k\in J'_{i_0}} \delta'_{i_0 k} \leqslant_1 \tau_2$, and $J_i \subseteq K_i\ ]$, so *(ii)* is satisfied. □

Lemma 3.11
If $u \leqslant_1 \bigvee_{i\in I} \delta_i$, then there exists $i_0 \in I$ such that $u \leqslant_1 \delta_{i_0}$.

Proof
From Lemma 3.10 we have that there are two cases.

If $\bigvee_{i\in I} \delta_i = u \vee \delta'$, we have $i_0 \in I$ such that $u \leqslant_1 \delta_{i_0}$.

If $u = \bigwedge_{k\in K}(\sigma_k \to \sigma'_k)$ and $\bigvee_{i\in I} \delta_i = (\tau_1 \to \tau_2) \vee \tau'$, and $\exists J : [\ \tau_1 \leqslant_1 \bigvee_{k\in J} \sigma_k$ and $\bigvee_{k\in J} \sigma'_k \leqslant_1 \tau_2$ and $J \subseteq K\ ]$, then choose $i_0 \in I$ such that $\tau_1 \to \tau_2 \leqslant_1 \delta_{i_0}$. From Lemma 3.10 we have that $u \leqslant_1 \tau_1 \to \tau_2$. From Rule (16), $u \leqslant_1 \tau_1 \to \tau_2$, and $\tau_1 \to \tau_2 \leqslant_1 \delta_{i_0}$, derive $u \leqslant_1 \delta_{i_0}$. □

Theorem 3.12
The relation $\leqslant_1$ is an acceptable type ordering.

Proof
We will prove that each of the five conditions from Definition 3.2 are satisfied.

1. The relation $\leqslant_1$ is reflexive because $\tau \leqslant_1 \tau \vee \bot = \tau$.
2. The relation $\leqslant_1$ is transitive by Rule (16).
3. We must show that if $\bigwedge_{k\in K}(\sigma_k \to \sigma'_k) \leqslant_1 (\tau_1 \to \tau_2)$, and $u \leqslant_1 \tau_1$, then there exists $k_0 \in K$ such that $u \leqslant_1 \sigma_{k_0}$ and $\sigma'_{k_0} \leqslant_1 \tau_2$. From Lemma 3.10 we have that there are two cases:
   (a) If $\tau_1 \to \tau_2 = (\bigwedge_{k\in K}(\sigma_k \to \sigma'_k)) \vee \tau'$, then $\tau_1 \to \tau_2 = (\bigwedge_{k\in K}(\sigma_k \to \sigma'_k))$. Choose $k_0 \in K$ such that $\tau_1 \to \tau_2 = \sigma_{k_0} \to \sigma'_{k_0}$. We have $u \leqslant_1 \tau_1 = \sigma_{k_0}$ and $\sigma'_{k_0} = \tau_2$.
   (b) If $\exists J : \tau_1 \leqslant_1 \bigvee_{k\in J} \sigma_k$ and $\bigvee_{k\in J} \sigma'_k \leqslant_1 \tau_2$ and $J \subseteq K$, then from Lemma 3.11 we have $k_0 \in J$ such that $u \leqslant_1 \sigma_{k_0}$. Moreover, $\sigma'_{k_0} \leqslant_1 \bigvee_{k\in J} \sigma'_k \leqslant_1 \tau_2$.
   (c) We have $\bigwedge_{k\in K}(\sigma_k \to \tau_k) \not\leqslant_1 \mathsf{Int}$ from Lemma 3.10.
4. We have $\mathsf{Int} \not\leqslant_1 \sigma \to \tau$ from Lemma 3.10.

□

Corollary 3.13
A program typable in $\mathscr{T}_{\leqslant_1}$ cannot go wrong.

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

Proof
Combine Corollary 3.9 and Theorem 3.12. □

Notice that (i) there is no separate rule for comparing recursive types, (ii) $\bot$ is a least type in the $\leqslant_1$-ordering because $\bot \leqslant_1 \bot \vee \tau = \tau$, (iii) if $\tau \leqslant_1 \bot$, then $\tau = \bot$, and (iv) there is no greatest type $\top$. Observation (iii) is straightforward to prove by induction on the structure of the derivation of $\tau \leqslant_1 \bot$ (and it also follows from Lemma 3.10).

In general, an expression does not have a $\leqslant_1$-minimal type in $\mathscr{T}_{\leqslant_1}$. For example, we have

$$\mathscr{T}_{\leqslant_1} \triangleright \emptyset \vdash (\lambda x.x^1)^2 : \mathsf{Int} \to \mathsf{Int}$$
$$\mathscr{T}_{\leqslant_1} \triangleright \emptyset \vdash (\lambda x.x^1)^2 : \bot \to \bot.$$

The types $\mathsf{Int} \to \mathsf{Int}$ and $\bot \to \bot$ have a greatest lower bound in $\leqslant_1$ which is $\mathsf{Int} \to \bot$, but we do not have $\mathscr{T}_{\leqslant_1} \triangleright \emptyset \vdash (\lambda x.x^1)^2 : \mathsf{Int} \to \bot$.

4 Polyvariant flow analysis

4.1 Domains

We begin by defining domains for the flow analysis. We use E to range over closed expressions. Our domains are parameterized by an expression E:

$$\begin{array}{lcll}
a & \in & \mathit{Val}(E) & = \mathit{Closure}(E) \cup \{\mathsf{Int}\} \quad \text{(abstract values)} \\
s & \in & \mathit{ValSet}(E) & = \mathscr{P}(\mathit{Val}(E)) \\
 & & \mathit{Exp}(E) & = \{\, e \in \mathit{Exp} \mid e \text{ occurs in } E \,\} \\
 & & \mathit{Abs}(E) & = \{\, (\lambda x.e)^l \in \mathit{Exp} \mid (\lambda x.e)^l \text{ occurs in } E \,\} \\
 & & \mathit{Var}(E) & = \{\, x \in \mathit{Var} \mid \exists e, l : (\lambda x.e)^l \text{ occurs in } E \,\} \\
 & & \mathit{Closure}(E) & = \mathit{Abs}(E) \times \mathit{FlowEnv}(E) \text{ (abstract closures)} \\
\rho & \in & \mathit{FlowEnv}(E) & = \mathit{Var}(E) \hookrightarrow \mathit{ValSet}(E) \\
 & & \mathit{FlowJudgment}(E) & = \mathit{FlowEnv}(E) \times \mathit{Exp}(E) \times \mathit{ValSet}(E) \\
R & \in & \mathit{FlowJudgmentSet}(E) & = \mathscr{P}(\mathit{FlowJudgment}(E)) \\
C & \in & \mathit{Cover}(E) & = \mathscr{P}(\mathit{ValSet}(E)).
\end{array}$$

An abstract value $a \in \mathit{Val}(E)$ is either an abstract closure or the constant Int. Our flow sets are sets of abstract values, that is, elements of $\mathit{ValSet}(E)$. The function $\mathscr{P}$ maps a set to its powerset. An abstract closure is an abstraction of a usual closure. A usual closure is a pair of a λ-abstraction and a static environment. An abstract closure contains a flow environment in place of the static environment. We will write elements of products as (x, y), etc., except for type judgments which we write as $A \vdash e : \tau$, as seen earlier, and elements of $\mathit{Closure}(E)$ which we write as $\langle (\lambda x.e)^l, \rho \rangle$, etc., for readability. A flow environment is a partial function with finite domain from $\mathit{Var}(E)$ to $\mathit{ValSet}(E)$. We use $\to$ to denote the constructor of spaces of total functions. (We also use $\to$ to denote the function-type constructor, but the intended meaning of $\to$ will always be clear from the context.)

A flow judgment (ρ, e, s) indicates that in the flow environment ρ, the expression e abstractly evaluates to the flow set s. Below, we define the valid flow judgments. Let us compare a flow judgment (ρ, e, s) to a type judgment $A \vdash e : \tau$. The flow

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

environment ρ and the type environment A play analogous roles. Similarly, the flow set s and the type τ play analogous roles.

4.2 The family of analyses

We define the set $Valid(E)$ of valid flow judgments as the greatest fixed point of the function F defined below. The interesting case is that of function application. At every call site, and for every function that can be invoked at that call site, we want to *cover* the set s_2 of abstract values for the actual argument. Thus, the cover is a set of sets of abstract values, and the union of the sets in the cover must contain s_2. Moreover, each function body must be analyzed as many times as the number of elements in the cover.

$$
\begin{array}{rcl}
F & \in & FlowJudgmentSet(E) \to FlowJudgmentSet(E) \\
F(R) & = & \{\ (\rho[x : s'], x^l, s)\ \mid\ s' \subseteq s\ \} \\
 & \cup & \{\ (\rho, (\lambda x.e)^l, s)\ \mid\ \langle (\lambda x.e)^l, \rho\rangle \in s\ \} \\
 & \cup & \{\ (\rho, (e_1\ e_2)^l, s)\ \mid \\
 & & \quad \exists s_1, s_2 : (\rho, e_1, s_1), (\rho, e_2, s_2) \in R \text{ and} \\
 & & \quad\quad \forall \langle (\lambda x.e)^{l'}, \rho'\rangle \in s_1 : \exists C \in Cover(E) : \\
 & & \quad\quad\quad s_2 \subseteq \bigcup C \text{ and} \\
 & & \quad\quad\quad \forall s' \in C : \exists s'' : (\rho'[x : s'], e, s'') \in R \text{ and } s'' \subseteq s\ \} \\
 & \cup & \{\ (\rho, c^l, s)\ \mid\ \mathsf{Int} \in s\ \} \\
 & \cup & \{\ (\rho, (\mathsf{succ}\ e_1)^l, s)\ \mid\ \mathsf{Int} \in s \text{ and } \exists s_1 : (\rho, e_1, s_1) \in R\ \} \\
 & \cup & \{\ (\rho, (\mathsf{if0}\ e_1\ e_2\ e_3)^l, s)\ \mid \\
 & & \quad \exists s_1, s_2, s_3 : (\rho, e_1, s_1), (\rho, e_2, s_2), (\rho, e_3, s_3) \in R \text{ and} \\
 & & \quad\quad s_2 \subseteq s \text{ and } s_3 \subseteq s\ \}
\end{array}
$$

We leave implicit that all the expressions mentioned in the definition of F must occur in E. The notation $\bigcup C$ means "the union of the sets that are elements of C." The cases for $x^l, (\lambda x.e)^l, c^l$ serve as "base cases" for the definition of F, as they do not refer to R.

The case for $(\lambda x.e)^l$ indicates that our closures are not "flat" in the sense of Appel (1992): in a closure $\langle (\lambda x.e)^l, \rho\rangle$, we can have that ρ is defined on a variable which does not occur free in $(\lambda x.e)^l$. Our choice of definition helps simplify the algorithm which translates type information into flow information, see Section 5.3.

The case for $(\mathsf{if0}\ e_1\ e_2\ e_3)^l$ indicates that the analysis does not attempt to statically decide which branch will be taken. The type system in section 3 does not attempt to do that either. One might change the flow analysis to have more fine-grained information about integers, and then attempt to decide if a condition will always (or never) evaluate to 0. To change the type system in a similar way, one possibility is to use the conditional types of Aiken *et al.* (1994).

All six cases in the definition of F allow the resulting flow set s to be larger than strictly necessary. Thus, one can view the rules as having a form of 'subsumption' built in, as expressed by the following lemma. As a consequence, there are many

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

valid flow judgments for a given expression, just as there can be many valid type judgments for a given expression.

Lemma 4.1
(Flow Subsumption) For $R \in \mathit{FlowJudgmentSet}(E)$, if $(\rho, e, s) \in F(R)$ and $s \subseteq s'$, then $(\rho, e, s') \in F(R)$.

Proof
By a straightforward case analysis on (ρ, e, s). □

Notice that $\mathit{FlowJudgmentSet}(E)$ with $\subseteq$ as ordering is a complete lattice. It is straightforward to show that F is *monotone*, that is, if $R_1 \subseteq R_2$, then $F(R_1) \subseteq F(R_2)$. From Tarski's fixed-point theorem (Tarski, 1955) we have that F has a greatest fixed point which we will denote by $\mathit{Valid}(E)$.

Lemma 4.2
If $R \subseteq F(R)$, then $R \subseteq \mathit{Valid}(E)$.

Proof
From Tarski's fixed point theorem we have that

$$\mathit{Valid}(E) = \bigcup \{\, R \mid R \subseteq F(R) \,\}$$

so if $R \subseteq F(R)$, then $R \subseteq \mathit{Valid}(E)$. □

Lemma 4.2 justifies the proof technique of co-induction (Milner and Tofte, 1991) which we will use repeatedly:

Proof by co-induction: Suppose $R \subseteq \mathit{FlowJudgment}(E)$. To prove $R \subseteq \mathit{Valid}(E)$, it is sufficient to prove $R \subseteq F(R)$.

We have already seen that a flow judgment (ρ, e, s) and a type judgment $A \vdash e : \tau$ play analogous roles.

Question: What plays a role analogous to a type *derivation* of a type judgment $A \vdash e : \tau$?

It should be a set of valid flow judgments that contains at least one judgment for e. Moreover, just like a type derivation can be checked independently of other type derivations, we want to be able to check the validity of the set of flow judgments independently of other flow judgments. Such independent checking can be done by co-induction, and we arrive at the following definition.

Definition 4.3
We say that $R \in \mathit{FlowJudgmentSet}(E)$ is an F-analysis of an expression e in a program E if and only if $R \subseteq F(R)$ and $\exists \rho : \exists s : (\rho, e, s) \in R$.

Given an F-analysis R of E, Lemma 4.2 shows that $R \subseteq \mathit{Valid}(E)$. In section 5 we will define mappings from F-analyses to type derivations, and back.

The self-contained nature of an F-analysis is largely made possible by the use of sets of flow judgments and 'local' environments that need not be defined on all bound variables in the whole program. In contrast, Nielson and Nielson (1997) use a cache of flow information for the whole program (rather than sets of flow judgments), and 'global' (rather than 'local') environments.

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

In the introduction we discussed how various analyses can be understood in terms of how they cover the flow set for an actual argument. In the following two cases, we can make the intuition precise by specializing the definition of F. It is done by, in the case for $(e_1 e_2)^l$, after '$\exists C \in \mathit{Cover}(E)$', inserting an extra condition on C:

- **Schmidt's analysis.** $C = \{s_2\}$.
- **Agesen's analysis.** $C = \{\ \{a\}\ \mid\ a \in s_2\ \}$.

We say that an F-analysis R of E is a 0-CFA-style analysis if and only if R satisfies that there exists $\rho \in \mathit{FlowEnv}(E)$ such that:

- if $\langle (\lambda y.e)^l, \rho' \rangle \in \rho(x)$, then $\rho' = \rho$;
- if $(\rho', e, s) \in R$, then $\rho' = \rho$;
- if $(\rho, e, s) \in R$ and $\langle (\lambda y.e')^l, \rho' \rangle \in s$, then $\rho' = \rho$; and
- if $(\rho, e, s), (\rho, e, s') \in R$, then $s = s'$.

These conditions express that a unique environment ρ is the only one used in R, and that there is a single judgment for each expression e in E. The conditions entail that each cover consists of a single set. To see that, notice that in the case for $(e_1 e_2)^l$ in the definition of F, if $\langle (\lambda x.e)^{l'}, \rho \rangle \in s_1$, then $C = \{\rho(x)\}$ and $s_2 \subseteq \rho(x)$.

For some programs, the least and the greatest fixed points of F are different. For example, consider the following λ-term and definitions. To distinguish environments, we will use numbers as subscripts. We will choose the numbers to be different from the labels of expressions.

$$
\begin{array}{rcl}
E & = & ((\lambda x.(x^1\ x^2)^3)^4\ (\lambda y.(y^5\ y^6)^7)^8)^9 \\
a_{\lambda x} & = & \langle (\lambda x.(x^1\ x^2)^3)^4, \emptyset \rangle \\
a_{\lambda y} & = & \langle (\lambda y.(y^5\ y^6)^7)^8, \emptyset \rangle \\
\rho_{20} & = & \emptyset[x : \{a_{\lambda y}\}] \\
\rho_{21} & = & \emptyset[y : \{a_{\lambda y}\}] \\
R & = & \{\ (\rho_{20}, x^1, \{a_{\lambda y}\}),\ (\rho_{20}, x^2, \{a_{\lambda y}\}), \\
 & & \ (\rho_{20}, (x^1\ x^2)^3, \emptyset),\ (\emptyset, (\lambda x.(x^1\ x^2)^3)^4, \{a_{\lambda x}\}), \\
 & & \ (\rho_{21}, y^5, \{a_{\lambda y}\}),\ (\rho_{21}, y^6, \{a_{\lambda y}\}), \\
 & & \ (\rho_{21}, (y^5\ y^6)^7, \emptyset),\ (\emptyset, (\lambda y.(y^5\ y^6)^7)^8, \{a_{\lambda y}\}), \\
 & & \ (\emptyset, E, \emptyset)\ \}
\end{array}
$$

It is straightforward to show $R \subseteq F(R)$ by case analysis of the elements of R, so R is an F-analysis of E. Define

$$
\begin{array}{rcl}
R' & = & \{\ (\rho[x : s'], x^1, s)\ \mid\ s' \subseteq s\ \} \\
 & \cup & \{\ (\rho[x : s'], x^2, s)\ \mid\ s' \subseteq s\ \} \\
 & \cup & \{\ (\rho, (\lambda x.(x^1\ x^2)^3)^4, s)\ \mid\ \langle (\lambda x.(x^1\ x^2)^3)^4, \rho \rangle \in s\ \} \\
 & \cup & \{\ (\rho[y : s'], y^5, s)\ \mid\ s' \subseteq s\ \} \\
 & \cup & \{\ (\rho[y : s'], y^6, s)\ \mid\ s' \subseteq s\ \} \\
 & \cup & \{\ (\rho, (\lambda y.(y^5\ y^6)^7)^8, s)\ \mid\ \langle (\lambda y.(y^5\ y^6)^7)^8, \rho \rangle \in s\ \}
\end{array}
$$

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

It is straightforward to show $F(\emptyset) = R'$ and $F(R') = R'$, so R' is the least fixed point of F. Notice that there are no flow judgments in R' for applications, while, for example, $(\emptyset, E, \emptyset) \in R \subseteq \mathit{Valid}(E)$. We conclude that $R' \neq \mathit{Valid}(E)$, that is, for E the least and the greatest fixed points of F are different.

Nielson and Nielson (1997) also gave an example, in the setting of a different flow analysis, of how the least and greatest fixed points can be different. In our paper, the main result, which relates the flow analysis to a type system, relies on that $\mathit{Valid}(E)$ is defined as the *greatest* fixed point of F.

As mentioned in the introduction, flow preservation does *not* hold:

It is false that if $(\rho, e, s) \in \mathit{Valid}(E)$ and $e \rightarrow_V e'$, then $(\rho, e', s) \in \mathit{Valid}(E)$.

For example, consider the program E and the definitions of $\rho, a_{\lambda y}, a_{\lambda x}, R$:

$$\begin{array}{rcl} E & = & ((\lambda x.(\lambda y.x^1)^2)^3\ 8^4)^5 \\ \rho & = & \emptyset[x : \{\mathsf{Int}\}] \\ a_{\lambda y} & = & \langle (\lambda y.x^1)^2, \rho \rangle \\ a_{\lambda x} & = & \langle (\lambda x.(\lambda y.x^1)^2)^3, \emptyset \rangle \\ R & = & \{\ (\emptyset, E, \{a_{\lambda y}\}), (\emptyset, (\lambda x.(\lambda y.x^1)^2)^3, \{a_{\lambda x}\}), \\ & & \quad (\emptyset, 8^4, \{\mathsf{Int}\}), (\rho, (\lambda y.x^1)^2, \{a_{\lambda y}\})\ \} \end{array}$$

It is straightforward to show $R \subseteq F(R)$ by case analysis of the elements of R, so R is an F-analysis of E. We have $(\emptyset, E, \{a_{\lambda y}\}) \in \mathit{Valid}(E)$ and $E \rightarrow_V (\lambda y.8^4)^2$, but $(\emptyset, (\lambda y.8^4)^2, \{a_{\lambda y}\}) \notin \mathit{Valid}(E)$.

We will present a more coarse-grained correctness result in section 5 (Corollary 5.2). Moreover, the flow-type system that we will study in section 7 embodies the ideas of the flow analysis from this section, and for that flow-type system we do have flow-type preservation and flow-type soundness.

4.3 Example

We now continue the example from section 1 by presenting an Agesen-style analysis of the following program:

$$E = ((\lambda f.(\mathsf{succ}\ ((f^1 f^2)^3 \mathsf{0}^4)^5)^6)^7\ (\mathsf{if0}\ c\ (\lambda x.x^8)^9\ (\lambda y.(\lambda z.z^{10})^{11})^{12})^{13})^{14}.$$

Define

$$\begin{array}{rcl} a_{\lambda f} & = & \langle (\lambda f.(\mathsf{succ}\ ((f^1 f^2)^3 \mathsf{0}^4)^5)^6)^7, \emptyset \rangle \\ a_{\lambda x} & = & \langle (\lambda x.x^8)^9, \emptyset \rangle \\ a_{\lambda y} & = & \langle (\lambda y.(\lambda z.z^{10})^{11})^{12}, \emptyset \rangle \\ \rho_{20} & = & \emptyset[y : \{a_{\lambda y}\}] \\ a_{\lambda z} & = & \langle (\lambda z.z^{10})^{11}, \rho_{20} \rangle \\ \rho_{21} & = & \emptyset[f : \{a_{\lambda x}\}] \\ \rho_{22} & = & \emptyset[f : \{a_{\lambda y}\}] \\ \rho_{23} & = & \emptyset[x : \{\mathsf{Int}\}] \\ \rho_{24} & = & \emptyset[x : \{a_{\lambda x}\}] \end{array}$$

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

$$
\begin{array}{rcl}
\rho_{25} & = & \emptyset[y : \{a_{\lambda y}\}]\ [z : \{\mathsf{Int}\}] \\
R & = & \{\ (\rho_{21}, f^1, \{\ a_{\lambda x}\ \}),\ (\rho_{21}, f^2, \{\ a_{\lambda x}\ \}),\ (\rho_{21}, (f^1 f^2)^3, \{\ a_{\lambda x}\ \}), \\
 & & \quad (\rho_{21}, 0^4, \{\ \mathsf{Int}\ \}),\ (\rho_{21}, ((f^1 f^2)^3 0^4)^5, \{\ \mathsf{Int}\ \}), \\
 & & \quad (\rho_{21}, (\mathsf{succ}\ ((f^1 f^2)^3 0^4)^5)^6, \{\ \mathsf{Int}\ \}), \\
 & & \quad (\rho_{22}, f^1, \{\ a_{\lambda y}\ \}),\ (\rho_{22}, f^2, \{\ a_{\lambda y}\ \}),\ (\rho_{22}, (f^1 f^2)^3, \{\ a_{\lambda z}\ \}), \\
 & & \quad (\rho_{22}, 0^4, \{\ \mathsf{Int}\ \}),\ (\rho_{22}, ((f^1 f^2)^3 0^4)^5, \{\ \mathsf{Int}\ \}), \\
 & & \quad (\rho_{22}, (\mathsf{succ}\ ((f^1 f^2)^3 0^4)^5)^6, \{\ \mathsf{Int}\ \}), \\
 & & \quad (\emptyset, (\lambda f.(\mathsf{succ}\ ((f^1 f^2)^3 0^4)^5)^6)^7, \{\ a_{\lambda f}\ \}), \\
 & & \quad (\rho_{23}, x^8, \{\ \mathsf{Int}\ \}),\ (\rho_{24}, x^8, \{\ a_{\lambda x}\ \}),\ (\emptyset, (\lambda x.x^8)^9, \{\ a_{\lambda x}\ \}), \\
 & & \quad (\rho_{25}, z^{10}, \{\ \mathsf{Int}\ \}),\ (\rho_{20}, (\lambda z.z^{10})^{11}, \{\ a_{\lambda z}\ \}), \\
 & & \quad (\emptyset, (\lambda y.(\lambda z.z^{10})^{11})^{12}, \{\ a_{\lambda y}\ \}), \\
 & & \quad (\emptyset, (\mathsf{if0}\ c\ (\lambda x.x^8)^9\ (\lambda y.(\lambda z.z^{10})^{11})^{12})^{13}, \{\ a_{\lambda x}, a_{\lambda y}\}), \\
 & & \quad (\emptyset, E, \{\ \mathsf{Int}\ \}), \\
 & & \}
\end{array}
$$

It is straightforward to show $R \subseteq F(R)$ by case analysis of the elements of R, so R is an F-analysis of E.

5 Equivalence

In this section we prove our main result which is the following theorem. The statement of the theorem uses the concept of *F-flow-safe* which is defined in section 5.1.

Theorem 5.1
(Main Theorem) A program is typable in $\mathscr{T}_{\leqslant_1}$ if and only if it is F-flow-safe.

Proof
The two implications are given by Theorems 5.6 and 5.8. □

Corollary 5.2
If a program is F-flow-safe, then it cannot go wrong.

Proof
Combine Corollary 3.13 and Theorem 5.1. □

5.1 Basic definitions

For the purpose of proving equivalences between type systems and flow analyses we will need the following definitions.

$$
\begin{array}{rcl}
collect & \in & FlowJudgmentSet(E) \to ValSet(E) \\
collect(R) & = & \{\ a \in Val(E)\ \mid \\
 & & \quad (\rho, e, s) \in R \text{ and either} \\
 & & \quad\quad - \ a \in subtrees(\rho(x)) \text{ and } x \in dom(\rho), \text{ or} \\
 & & \quad\quad - \ a \in subtrees(s)\ \}
\end{array}
$$

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

$$subtrees \quad \in \quad ValSet(E) \to ValSet(E)$$

subtrees is the pointwise $\subseteq$-smallest function such that

$$\begin{aligned} subtrees(s) \quad = \quad & \{\, a \in Val(E) \mid \\ & \quad \text{either } a \in s, \\ & \quad \text{or } \exists \langle (\lambda x.e)^l, \rho \rangle \in s : \exists y \in dom(\rho) : a \in subtrees(\rho(y)) \,\} \end{aligned}$$

$$argres \quad \in \quad (Closure(E) \times FlowJudgmentSet(E)) \to \mathscr{P}(ValSet(E) \times ValSet(E))$$

$$argres(\langle (\lambda x.e)^l, \rho \rangle, R) = \{\, (s', s'') \mid (\rho[x : s'], e, s'') \in R \,\}$$

Intuitively, $collect(R)$ is the set of abstract values involved in R. To capture that an abstract value can be part of the environment of another abstract value, we use the function *subtrees*. We use $argres(a, R)$ to model the argument-result behavior of a with respect to R.

For $R \in FlowJudgmentSet(E)$, define

- R is *safe* if and only if
  - — if $(e_1\ e_2)^l \in Exp(E)$, and $(\rho, e_1, s) \in R$, then $\mathsf{Int} \notin s$;
  - — if $(\mathsf{succ}\ e_1)^l \in Exp(E)$, and $(\rho, e_1, s) \in R$, then $s \subseteq \{\mathsf{Int}\}$;
  - — if $(\mathsf{if0}\ e_1\ e_2\ e_3)^l \in Exp(E)$, and $(\rho, e_1, s) \in R$, then $s \subseteq \{\mathsf{Int}\}$
- R is *finitary* if and only if $collect(R)$ is a finite set
- R *analyzes all its closure bodies* if and only if $\forall a \in collect(R) \cap Closure(E) : argres(a, R) \neq \emptyset$.

The conditions for R being safe correspond to the safety checks of (Palsberg and Schwartzbach, 1995; Palsberg and O'Keefe, 1995).

Note that R can be finite without being finitary because an element of R may contain infinitely many distinct subtrees. A 0-CFA-style analysis is always finitary, while an Agesen-style analysis need not be finitary (Agesen, 1995b). For Schmidt's analysis, it remains open whether it is always finitary.

Intuitively, we have the following correspondences:

Flows	*Types*
safe	type safe
finitary	all intersection and union types have a finite number of components
analyzes all closure bodies	all intersection types are nonempty that is, even 'dead code' must be well typed.

We can now define the key notion of a program being F-flow-safe.

Definition 5.3
We say that E is F-flow-safe if and only if there exists an F-analysis R of E, such that R is safe, finitary and analyzes all its closure bodies.

We will need the following observation to ensure that certain sets are finite.

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

Lemma 5.4
For $R \in FlowJudgmentSet(E)$, if R is finitary and $a \in collect(R) \cap Closure(E)$, then $argres(a, R)$ is a finite set.

Proof
Notice that

$$argres(a, R) \subseteq \mathscr{P}(collect(R) \times collect(R)),$$

and that $collect(R)$ is a finite set by assumption. □

Define $(f \circ g)(x) = f(g(x))$. We will use the notational convention that $f \circ g(x)$ should be grouped as $(f \circ g)(x)$.

5.2 *From flows to types*

Given $R \in FlowJudgmentSet(E)$, where R is finitary and analyzes all its closure bodies, we define a system of type equations $TypeEqSys(R)$:

- **Type variables:** in $TypeEqSys(R)$ the set of type variables is denoted by $\mathscr{W}(R)$ and it consists of one type variable W_a for each $a \in collect(R) \cap Closure(E)$. Notice that $\mathscr{W}(R)$ is a finite set because R is finitary. We assume that $\mathscr{W}(R) \subseteq TypeVar$, where $TypeVar$ is the set of type variables from section 3.
- **Auxiliary function:** define

$$\begin{array}{rcl} \tau & \in & EquationType(R) \\ \tau & ::= & \mathsf{Int} \mid \bot \mid \tau \vee \tau' \mid W \end{array}$$

  where W ranges over $\mathscr{W}(R)$. Define

$$\begin{array}{rcl} f & : & collect(R) \to EquationType(R) \\ f(\emptyset) & = & \bot \\ f(\{\mathsf{Int}\}) & = & \mathsf{Int} \\ f(\{a\}) & = & W_a \quad (a \in Closure(E)) \\ f(s_1 \cup s_2) & = & f(s_1) \vee f(s_2). \end{array}$$

  Notice that $f(s)$ is a finite disjunction because R is finitary. For example,

$$f(\{\ \langle(\lambda x.x^8)^9, \emptyset\rangle,\ \langle(\lambda y.(\lambda z.z^{10})^{11})^{12}, \emptyset\rangle\ \}) = W_{\langle(\lambda x.x^8)^9,\emptyset\rangle} \vee W_{\langle(\lambda y.(\lambda z.z^{10})^{11})^{12},\emptyset\rangle}.$$

- **Equations:** for each $W_a \in \mathscr{W}(R)$, $TypeEqSys(R)$ contains the equation

$$W_a = \bigwedge_{(s',s'') \in argres(a,R)} (f(s') \to f(s'')), \tag{22}$$

  where the two applications of f should be replaced by their results to obtain the actual equation. From Lemma 5.4 and since R is finitary and analyzes all its closure bodies, we have that the conjunction is nonempty and finite. Since $\mathscr{W}(R)$ is a finite set, there are finitely many equations.
- **Solutions:** A solution of $TypeEqSys(E)$ is a mapping

$$\psi : \mathscr{W}(R) \to Type$$

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

such that if we have an equation of the form (22), then

$$\psi^\bullet(W_a) = \bigwedge_{(s',s'')\in argres(a,R)} ((\psi^\bullet \circ f(s')) \to (\psi^\bullet \circ f(s'')))$$

where

$$\psi^\bullet : EquationType(R) \to Type$$

is the unique extension of ψ to a type-homomorphism:

$$\begin{aligned} \psi^\bullet(\bot) &= \bot \\ \psi^\bullet(\mathsf{Int}) &= \mathsf{Int} \\ \psi^\bullet(W_a) &= \psi(W_a) \\ \psi^\bullet(\tau_1 \vee \tau_2) &= \psi^\bullet(\tau_1) \vee \psi^\bullet(\tau_2). \end{aligned}$$

Unique solution: every equation system $TypeEqSys(R)$ has a unique solution (Courcelle, 1983). To see that, notice that for every variable in the equation system, there is exactly one equation with that variable as left-hand side. Moreover, on the right-hand sides of the equations, all type variables occur below at least one function type constructor. Thus, intuitively, we obtain the unique solution by using each equation as an unfolding rule, possibly infinitely often.

Lemma 5.5
The following are true:

1. If $s \subseteq s'$, then $f(s) \leqslant_1 f(s')$.
2. If $\tau \leqslant_1 \tau'$, then $\psi^\bullet(\tau) \leqslant_1 \psi^\bullet(\tau')$.
3. $\psi^\bullet \circ f \circ (\rho[x : s]) = (\psi^\bullet \circ f \circ \rho)[c : (\psi^\bullet \circ f)(s)]$.
4. If we can derive $A \vdash e : \psi^\bullet \circ f(s)$, and $s \subseteq s'$, then we can derive $A \vdash e : \psi^\bullet \circ f(s')$.

Proof
The properties (1), (2), (3) are immediate. For property 4, notice that $s \subseteq s'$ and the properties (1), (2) imply $\psi^\bullet \circ f(s) \leqslant_1 \psi^\bullet \circ f(s')$. So, from $A \vdash e : \psi^\bullet \circ f(s)$ and Rule (15), we can derive $A \vdash e : \psi^\bullet \circ f(s')$. □

Theorem 5.6
If E is F-flow-safe, then E is typable in $\mathscr{T}_{\leqslant_1}$.

Proof
From E being F-flow-safe we have an F-analysis R of E such that R is safe, finitary and analyzes all its closure bodies. Since R is finitary and analyzes all its closure bodies, we can construct $TypeEqSys(R)$. Let ψ be the unique solution of $TypeEqSys(R)$. We will show

$$\begin{aligned} &\forall e \in Exp(E) : \forall \rho, s : \\ &\quad \text{if } (\rho, e, s) \in R, \\ &\quad \text{then } \mathscr{T}_{\leqslant_1} \rhd \psi^\bullet \circ f \circ \rho \vdash e : \psi^\bullet \circ f(s). \end{aligned}$$

From R being an F-analysis of E we have that there exist ρ, s such that $(\rho, E, s) \in R$, so from the property just stated we have that E is typable in $\mathscr{T}_{\leqslant_1}$.

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

To prove the stated property, we proceed by induction on the structure of e. We will use repeatedly that since R is an analysis of E, then $R \subseteq F(R)$. Furthermore, we will use that $\mathsf{Int} = \psi^\bullet \circ f(\{\mathsf{Int}\})$. Finally, we will use that R is safe.

There are now six cases depending on e.

- $e \equiv x^l$. From $(\rho, x^l, s) \in R \subseteq F(R)$ we have $\rho \equiv \rho'[x : s']$ and $s' \subseteq s$. ¿From Lemma 5.5, item (3), we have $\psi^\bullet \circ f \circ \rho = (\psi^\bullet \circ f \circ \rho')[x : (\psi^\bullet \circ f)(s')]$, so we can derive
$$\frac{\psi^\bullet \circ f \circ \rho \vdash x^l : \psi^\bullet \circ f(s')}{\psi^\bullet \circ f \circ \rho \vdash x^l : \psi^\bullet \circ f(s)}$$
using Rule (9) and Lemma 5.5, item (4).
- $e \equiv (\lambda x.e_1)^l$. From $(\rho, (\lambda x.e_1)^l, s) \in R \subseteq F(R)$ we have $a \in s$ where $a = \langle (\lambda x.e_1)^l, \rho \rangle$, so $a \in \mathit{collect}(R) \cap \mathit{Closure}(E)$. Since R analyzes all its closure bodies, we have $\mathit{argres}(a, R) \neq \emptyset$, and for all $(s', s'') \in \mathit{argres}(a, R)$, we have $(\rho[x : s'], e_1, s'') \in R$. From the induction hypothesis we have that we can derive, for every $(s', s'') \in \mathit{argres}(a, R)$,
$$\psi^\bullet \circ f \circ (\rho[x : s']) \vdash e_1 : \psi^\bullet \circ f(s'').$$
This can also be written
$$(\psi^\bullet \circ f \circ \rho)[x : \psi^\bullet \circ f(s')] \vdash e_1 : \psi^\bullet \circ f(s'').$$
Derive
$$\psi^\bullet \circ f \circ \rho \vdash (\lambda x.e_1)^l : \bigwedge_{(s',s'') \in \mathit{argres}(a,R)} (\psi^\bullet \circ f(s')) \to (\psi^\bullet \circ f(s''))$$
using Rule (10). In $\mathit{TypeEqSys}(R)$ we have
$$W_a = \bigwedge_{(s',s'') \in \mathit{argres}(a,R)} (f(s') \to f(s'')).$$
so we can rewrite the previous type judgment as
$$\psi^\bullet \circ f \circ \rho \vdash (\lambda x.e_1)^l : \psi^\bullet(W_a).$$
Derive
$$\psi^\bullet \circ f \circ \rho \vdash (\lambda x.e_1)^l : \psi^\bullet \circ f(s)$$
using $a \in s$ and Lemma 5.5, item (4).
- $e \equiv (e_1 e_2)^l$. From $(\rho, (e_1 e_2)^l, s) \in R \subseteq F(R)$ we have s_1, s_2 such that
$$\begin{aligned}
&(\rho, e_1, s_1), (\rho, e_2, s_2) \in R \text{ and} \\
&\forall \langle (\lambda x.e_3)^{l'}, \rho' \rangle \in s_1 : \exists C \in \mathit{Cover}(E) : \\
&\qquad s_2 \subseteq \bigcup C \text{ and} \\
&\qquad \forall s' \in C : \exists s'' : (\rho'[x : s'], e_3, s'') \in R \text{ and } s'' \subseteq s
\end{aligned}$$
Since R is safe we have $\mathsf{Int} \notin s_1$. From the induction hypothesis we have that we can derive
$$\begin{aligned}
&\psi^\bullet \circ f \circ \rho \vdash e_1 : \psi^\bullet \circ f(s_1) \\
&\psi^\bullet \circ f \circ \rho \vdash e_2 : \psi^\bullet \circ f(s_2).
\end{aligned}$$

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

Choose an index set I such that $\{\ a_i\ \mid\ i \in I\ \} = s_1 \subseteq \mathit{Closure}(E)$, and write $a_i = \langle(\lambda x.e_i')^{l_i}, \rho_i\rangle$ for all $i \in I$. For all $i \in I$, let K_i be an index set such that, for $k \in K_i$, (s_{ik}', s_{ik}'') are the elements of $\mathit{argres}(a_i, R)$. For all $i \in I$, choose $C_i \in \mathit{Cover}(E)$ such that

$$s_2 \subseteq \bigcup C_i \text{ and}$$
$$\forall s' \in C_i : \exists s'' : (\rho_i[x : s'], e_i', s'') \in R \text{ and } s'' \subseteq s.$$

For all $i \in I$, we have

$$C_i \quad \subseteq \quad \{\ s_{ik}' \mid (s_{ik}', s_{ik}'') \in \mathit{argres}(a_i, R)\ \}$$

so choose J_i such that $J_i \subseteq K_i$ and $C_i = \{\ s_{ik}'\ \mid\ k \in J_i\ \}$. For all $i \in I$, we derive from the properties of C_i that

$$s_2 \quad \subseteq \quad \bigcup_{k \in J_i} s_{ik}'$$
$$\bigcup_{k \in J_i} s_{ik}'' \quad \subseteq \quad s.$$

Notice

$$\begin{aligned}
\psi^\bullet \circ f(s_1) &= \psi^\bullet \left(\bigvee_{i \in I} W_{a_i} \right) \\
&= \bigvee_{i \in I} \bigwedge_{(s', s'') \in \mathit{argres}(a_i, R)} ((\psi^\bullet \circ f(s')) \to (\psi^\bullet \circ f(s''))) \\
&= \bigvee_{i \in I} \bigwedge_{k \in K_i} ((\psi^\bullet \circ f(s_{ik}')) \to (\psi^\bullet \circ f(s_{ik}''))) \\
&\leqslant_1 (\psi^\bullet \circ f(s_2)) \to (\psi^\bullet \circ f(s))
\end{aligned}$$

where the inequality is obtained from Lemma 3.10 because for all $i \in I$ we have $\psi^\bullet \circ f(s_2) \leqslant_1 \bigvee_{k \in J_i} (\psi^\bullet \circ f(s_{ik}'))$ and $\bigvee_{k \in J_i} (\psi^\bullet \circ f(s_{ik}'')) \leqslant_1 \psi^\bullet \circ f(s)$ and $J_i \subseteq K_i$. Finally, derive

$$\psi^\bullet \circ f \circ \rho \vdash (e_1 e_2)^l : \psi^\bullet \circ f(s)$$

using Rules (15), (11).

- $e \equiv c^l$. From $(\rho, c^l, s) \in R \subseteq F(R)$ we have $\mathsf{Int} \in s$. Derive

$$\frac{\psi^\bullet \circ f \circ \rho \vdash c^l : \mathsf{Int}}{\psi^\bullet \circ f \circ \rho \vdash c^l : \psi^\bullet \circ f(s)}$$

using Rule (12) and Lemma 5.5, item (4).

- $e \equiv (\mathsf{succ}\ e_1)^l$. From $(\rho, (\mathsf{succ}\ e_1)^l, s) \in R \subseteq F(R)$ we have $\mathsf{Int} \in s$ and we have s_1 such that $(\rho, e_1, s_1) \in R$. From R being safe we have $s_1 \subseteq \{\mathsf{Int}\}$. From the induction hypothesis, Lemma 5.5, item (4), Rule (13), and again Lemma 5.5, item (4), derive

$$\frac{\dfrac{\dfrac{\psi^\bullet \circ f \circ \rho \vdash e_1 : \psi^\bullet \circ f(s_1)}{\psi^\bullet \circ f \circ \rho \vdash e_1 : \mathsf{Int}}}{\psi^\bullet \circ f \circ \rho \vdash (\mathsf{succ}\ e_1)^l : \mathsf{Int}}}{\psi^\bullet \circ f \circ \rho \vdash (\mathsf{succ}\ e_1)^l : \psi^\bullet \circ f(s)}.$$

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

- $e \equiv (\mathsf{if0}\ e_1\ e_2\ e_3)^l$. From $(\rho, (\mathsf{if0}\ e_1\ e_2\ e_3)^l, s) \in R \subseteq F(R)$ we have s_1, s_2, s_3 such that $(\rho, e_1, s_1), (\rho, e_2, s_2), (\rho, e_3, s_3) \in R$ and $s_2 \subseteq s$ and $s_3 \subseteq s$. From R being safe we have $s_1 \subseteq \{\mathsf{Int}\}$. The induction hypothesis provides derivations of

$$\begin{aligned}
\psi^\bullet \circ f \circ \rho &\vdash e_1 : \psi^\bullet \circ f(s_1) \\
\psi^\bullet \circ f \circ \rho &\vdash e_2 : \psi^\bullet \circ f(s_2) \\
\psi^\bullet \circ f \circ \rho &\vdash e_3 : \psi^\bullet \circ f(s_3),
\end{aligned}$$

and we can then use Lemma 5.5, item (4) to derive

$$\begin{aligned}
\psi^\bullet \circ f \circ \rho &\vdash e_1 : \mathsf{Int} \\
\psi^\bullet \circ f \circ \rho &\vdash e_2 : \psi^\bullet \circ f(s) \\
\psi^\bullet \circ f \circ \rho &\vdash e_3 : \psi^\bullet \circ f(s).
\end{aligned}$$

Finally use Rule (14) to derive

$$\psi^\bullet \circ f \circ \rho \vdash (\mathsf{if0}\ e_1\ e_2\ e_3)^l : \psi^\bullet \circ f(s).$$

□

We now complete the example from Section 1, using the R defined in section 4.3 which we have established is an F-analysis of E. It is straightforward to show that R is safe, finitary and analyzes all its closure bodies. Recall that

$$\begin{aligned}
a_{\lambda f} &= \langle (\lambda f.(\mathsf{succ}\ ((f^1 f^2)^3 \mathsf{0}^4)^5)^6)^7, \emptyset \rangle \\
a_{\lambda x} &= \langle (\lambda x.x^8)^9, \emptyset \rangle \\
a_{\lambda y} &= \langle (\lambda y.(\lambda z.z^{10})^{11})^{12}, \emptyset \rangle \\
\rho_{20} &= \emptyset[y : \{\langle (\lambda y.(\lambda z.z^{10})^{11})^{12}, \emptyset \rangle\}] \\
a_{\lambda z} &= \langle (\lambda z.z^{10})^{11}, \rho_{20} \rangle.
\end{aligned}$$

We have

$$\begin{aligned}
\mathit{collect}(R) &= \{\ a_{\lambda f}, a_{\lambda x}, a_{\lambda y}, a_{\lambda z}, \mathsf{Int}\ \} \\
\mathit{argres}(a_{\lambda f}, R) &= \{\ (\{a_{\lambda x}\}, \{\mathsf{Int}\}),\ (\{a_{\lambda y}\}, \{\mathsf{Int}\})\ \} \\
\mathit{argres}(a_{\lambda x}, R) &= \{\ (\{\mathsf{Int}\}, \{\mathsf{Int}\}),\ (\{a_{\lambda x}\}, \{a_{\lambda x}\})\ \} \\
\mathit{argres}(a_{\lambda y}, R) &= \{\ (\{a_{\lambda y}\}, \{a_{\lambda z}\})\ \} \\
\mathit{argres}(a_{\lambda z}, R) &= \{\ (\{\mathsf{Int}\}, \{\mathsf{Int}\}),\ \}
\end{aligned}$$

The equation system $\mathit{TypeEqSys}(R)$ contains the following four equations:

$$\begin{aligned}
W_{a_{\lambda f}} &= (W_{a_{\lambda x}} \to \mathsf{Int}) \wedge (W_{a_{\lambda y}} \to \mathsf{Int}) \\
W_{a_{\lambda x}} &= (\mathsf{Int} \to \mathsf{Int}) \wedge (W_{a_{\lambda x}} \to W_{a_{\lambda x}}) \\
W_{a_{\lambda y}} &= W_{a_{\lambda y}} \to W_{a_{\lambda z}} \\
W_{a_{\lambda z}} &= \mathsf{Int} \to \mathsf{Int}
\end{aligned}$$

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

Here is the unique solution ψ of $TypeEqSys(E)$:

$$\begin{aligned}\psi(W_{a_{\lambda f}}) &= (\sigma \to \mathsf{Int}) \wedge (\tau \to \mathsf{Int})\\ \psi(W_{a_{\lambda x}}) &= \sigma\\ \psi(W_{a_{\lambda y}}) &= \tau\\ \psi(W_{a_{\lambda z}}) &= \mathsf{Int} \to \mathsf{Int}\end{aligned}$$

where

$$\begin{aligned}\sigma &= \mu\alpha.((\mathsf{Int} \to \mathsf{Int}) \wedge (\alpha \to \alpha))\\ \tau &= \mu\beta.(\beta \to (\mathsf{Int} \to \mathsf{Int})).\end{aligned}$$

We invite the reader to revisit the example in section 1 to see again how σ and τ indeed yield a type derivation which shows $\mathscr{T}_{\leqslant_1} \triangleright \emptyset \vdash E : \mathsf{Int}$.

5.3 *From types to flows*

We use $TypeDerivation(\mathscr{T}_{\leqslant})$ to denote the set of type derivations that are possible in $\mathscr{T}_{\leqslant}$. Define

$$\begin{aligned}types &\in TypeDerivation(\mathscr{T}_{\leqslant}) \to \mathscr{P}(Type)\\ types(D) &= \{\, \sigma \mid \exists A, e, \tau : (A \vdash e : \tau) \text{ occurs in } D, \text{ and either}\\ &\qquad - \ \tau = \sigma \vee \tau', \text{ or}\\ &\qquad - \ A(x) = \sigma \vee \tau', \text{ and } x \in dom(A) \,\}\\ lamenv &\in (Type \times TypeDerivation(\mathscr{T}_{\leqslant})) \to \mathscr{P}(Abs \times TypeEnv)\\ lamenv(\tau, D) &= \{\, ((\lambda x.e)^l, A) \mid \exists \sigma : (A \vdash (\lambda x.e)^l : \sigma) \text{ occurs in } D \text{ and } \sigma \leqslant \tau \,\}.\end{aligned}$$

Given $D \in TypeDerivation(\mathscr{T}_{\leqslant})$, where the root of D is $\emptyset \vdash E : \tau$ for some type τ, we define a system of set equations $SetEqSys(D)$:

- **Set variables:** in $SetEqSys(D)$ the set of set variables ranging over $ValSet(E)$ is denoted by $\mathscr{Z}(D)$ and it consists of one set variable Z_T for each $T \in types(D) \cap IntersectionType$. Notice that $\mathscr{Z}(D)$ is a finite set because $types(D)$ is finite.
- **Auxiliary Function:** The function g maps types in $types(D)$ to set expressions:

$$\begin{aligned}g(\bot) &= \emptyset\\ g(\mathsf{Int}) &= \{\mathsf{Int}\}\\ g(T) &= Z_T \quad (T \in IntersectionType)\\ g(\tau_1 \vee \tau_2) &= g(\tau_1) \cup g(\tau_2).\end{aligned}$$

  Notice that $g(\sigma)$ is a finite union. For example,

$$g(\mathsf{Int} \vee (\mathsf{Int} \to \mathsf{Int})) = \{\mathsf{Int}\} \cup Z_{\mathsf{Int}\to\mathsf{Int}}.$$

- **Equations:** for each $T \in types(D) \cap IntersectionType$, $SetEqSys(D)$ contains the equation

$$Z_T = \bigcup_{((\lambda x.e)^l, A) \in lamenv(T,D)} \{\langle (\lambda x.e)^l, g \circ A \rangle\}, \tag{23}$$

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

where the occurrence of g should be applied to all types in A to obtain the actual equation. Notice that the union is finite, and that there are finitely many equations.

- **Solutions:** a solution of $SetEqSys(D)$ is a mapping φ from elements of $\mathscr{Z}(D)$ to elements of $ValSet(E)$ such that if we have equation of the form (23), then

$$\varphi^\bullet(Z_T) = \bigcup_{((\lambda x.e)^l, A) \in lamenv(T,D)} \{\langle (\lambda x.e)^l, \varphi^\bullet \circ g \circ A \rangle\}$$

where $\varphi^\bullet$ is the unique extension of φ to a set-expression-homomorphism:

$$\begin{aligned} \varphi^\bullet(\emptyset) &= \emptyset \\ \varphi^\bullet(\{\mathsf{Int}\}) &= \{\mathsf{Int}\} \\ \varphi^\bullet(Z_T) &= \varphi(Z_T) \\ \varphi^\bullet(s_1 \cup s_2) &= \varphi^\bullet(s_1) \cup \varphi^\bullet(s_2). \end{aligned}$$

Unique solution: every equation system $SetEqSys(D)$ has a unique solution. To see that, notice that for every variable in the equation system, there is exactly one equation with that variable as left-hand side. Moreover, on the right-hand sides of the equations, all set variables occur below at least one constructor. Thus, intuitively, we obtain the unique solution by using each equation as an unfolding rule, possibly infinitely often.

Lemma 5.7
Given $D \in TypeDerivation(\mathscr{T}_{\leqslant_1})$, and $\sigma, \tau \in types(D)$, let φ be the unique solution of $SetEqSys(D)$. If $\sigma \leqslant_1 \tau$, then $\varphi^\bullet \circ g(\sigma) \subseteq \varphi^\bullet \circ g(\tau)$.

Proof
From Lemma 3.10 we have that there are two cases:

- First, suppose $\tau = \sigma \vee \tau'$. From the definitions of $\varphi^\bullet$ and g we have

$$\begin{aligned} & \varphi^\bullet \circ g(\tau) \\ = \; & \varphi^\bullet \circ g(\sigma \vee \tau') \\ = \; & \varphi^\bullet(g(\sigma) \cup g(\tau')) \\ = \; & (\varphi^\bullet \circ g(\sigma)) \cup (\varphi^\bullet \circ g(\tau')) \\ \supseteq \; & \varphi^\bullet \circ g(\sigma). \end{aligned}$$

- Secondly, suppose $\sigma = \bigvee_{i \in I} \bigwedge_{k \in K_i} (\sigma_{ik} \to \sigma'_{ik})$, and $\tau = (\tau_1 \to \tau_2) \vee \tau'$, and $\forall i \in I : \exists J_i : [\, \tau_1 \leqslant_1 \bigvee_{k \in J_i} \sigma_{ik}$, and $\bigvee_{k \in J_i} \sigma'_{ik} \leqslant_1 \tau_2$, and $J_i \subseteq K_i \,]$. For all $i \in I$, define

$$T_i = \bigwedge_{k \in K_i} (\sigma_{ik} \to \sigma'_{ik}).$$

Notice that, for all $i \in I$, we have $T_i \in types(D) \cap IntersectionType$, and $T_i \leqslant_1 \tau_1 \to \tau_2$. We can now use the above together with the definitions of $\varphi^\bullet$ and g to do the following calculation. We will use that if $\sigma \leqslant_1 \tau$, then $lamenv(\sigma, D) \subseteq lamenv(\tau, D)$.

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

$$
\begin{aligned}
& \varphi^\bullet \circ g(\sigma) \\
= \;& \varphi^\bullet \circ g(\bigvee_{i\in I} \bigwedge_{k\in K_i} (\sigma_{ik} \to \sigma'_{ik})) \\
= \;& \varphi^\bullet \circ g(\bigvee_{i\in I} T_i) \\
= \;& \varphi^\bullet(\bigcup_{i\in I} g(T_i)) \\
= \;& \varphi^\bullet(\bigcup_{i\in I} Z_{T_i}) \\
= \;& \bigcup_{i\in I} \varphi^\bullet(Z_{T_i}) \\
= \;& \bigcup_{i\in I} \bigcup_{((\lambda x.e)^l, A)\in lamenv(T_i, D)} \{\langle (\lambda x.e)^l, \varphi^\bullet \circ g \circ A\rangle\} \\
\subseteq \;& \bigcup_{i\in I} \bigcup_{((\lambda x.e)^l, A)\in lamenv(\tau_1\to\tau_2, D)} \{\langle (\lambda x.e)^l, \varphi^\bullet \circ g \circ A\rangle\} \\
= \;& \bigcup_{i\in I} \varphi^\bullet(Z_{\tau_1\to\tau_2}) \\
= \;& \varphi^\bullet(Z_{\tau_1\to\tau_2}) \\
= \;& \varphi^\bullet \circ g(\tau_1 \to \tau_2) \\
\subseteq \;& (\varphi^\bullet \circ g(\tau_1 \to \tau_2)) \cup (\varphi^\bullet \circ g(\tau')) \\
= \;& \varphi^\bullet(g(\tau_1 \to \tau_2) \cup g(\tau')) \\
= \;& \varphi^\bullet \circ g((\tau_1 \to \tau_2) \vee g(\tau')) \\
= \;& \varphi^\bullet \circ g(\tau).
\end{aligned}
$$

□

Theorem 5.8
If E is typable in $\mathscr{T}_{\leqslant_1}$, then E is F-flow-safe.

Proof
From E being typable in $\mathscr{T}_{\leqslant_1}$ we have a canonical-form type derivation $D_E \in TypeDerivation(\mathscr{T}_{\leqslant_1})$ with root $\emptyset \vdash E : \tau_E$ for some type τ_E. Let φ be the unique solution of $SetEqSys(D_E)$. Define $R \in FlowJudgmentSet(E)$ such that

$$R \;=\; \{\, (\varphi^\bullet \circ g \circ A, e, \varphi^\bullet \circ g(\tau)) \;\mid\; (A \vdash e : \tau) \text{ occurs in } D_E \,\}$$

Notice that $(\emptyset, E, \varphi^\bullet \circ g(\tau_E)) \in R$.

We have

$$collect(R) \subseteq \{\mathsf{Int}\} \cup \left(\bigcup_{T\in types(D_E)\cap IntersectionType} \varphi^\bullet(Z_T) \right).$$

Notice that $types(D_E) \cap IntersectionType$ is finite, and that each $\varphi^\bullet(Z_T)$ is finite, so $collect(R)$ is finite, hence R is finitary. Notice also that for every $(\lambda x.e)^l \in Exp(E)$, we have in D_E:

$$\frac{\forall k \in K : A[x : \sigma_k] \vdash e : \tau_k}{A \vdash (\lambda x.e)^l : \bigwedge_{k\in K} (\sigma_k \to \tau_k)}$$

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

so $\langle (\lambda x.e)^l, \varphi^\bullet \circ g \circ A \rangle \in collect(R) \cap Closure(E)$, and, for all $k \in K$,

$$((\varphi^\bullet \circ g \circ A)[x : \varphi^\bullet \circ g(\sigma_k)], e, \varphi^\bullet \circ g(\tau_k)) \in R,$$

so $argres(\langle (\lambda x.e)^l, \varphi^\bullet \circ g \circ A \rangle, R) \neq \emptyset$, hence R analyzes all its closure bodies.

To show that R is safe, consider $(e_1\ e_2)^l \in Exp(E)$. Notice that every occurrence of a judgment in D_E with e_1 as the second component is of the form $A \vdash e_1 : \delta$, where $\delta \leqslant_1 \sigma \to \tau$, hence every judgment in R with e_1 as the second component is of the form

$$(\varphi^\bullet \circ g \circ A, e_1, \varphi^\bullet \circ g(\delta)).$$

From Lemma 3.10 and $\delta \leqslant_1 \sigma \to \tau$ we have that either $\delta = \sigma \to \tau$, or $\delta = \bigvee_{i \in I} T_i$, where, for all $i \in I$, $T_i \in IntersectionType$. From the definitions of $\varphi^\bullet$ and g, we have $\mathsf{Int} \notin \varphi^\bullet \circ g(\delta)$. Similar arguments can be given for occurrences of $(\mathsf{succ}\ e)^l$, $(\mathsf{if0}\ e_1\ e_2\ e_3)^l$, so we conclude that R is safe.

To show that E is F-flow-safe, it remains to be shown that $R \subseteq F(R)$. It is sufficient to show

$$\begin{aligned} &\forall D \in TypeDerivation(\mathscr{T}_{\leqslant_1}) : \forall A : \forall e : \forall \tau : \\ &\quad \text{if } D \text{ is a subtree of } D_E, \text{ and } D \text{ has root } A \vdash e : \tau, \\ &\quad \text{then } (\varphi^\bullet \circ g \circ A, e, \varphi^\bullet \circ g(\tau)) \in F(R). \end{aligned}$$

To prove this, we proceed by induction on the structure of D.

There are now seven subcases depending on which one of Rules (9)–(15) was the last one used in the derivation of $A \vdash e : \tau$.

- Rule (9). We have $e \equiv x^l$ and the derivation of $A \vdash e : \tau$ if of the form $A'[x : \tau] \vdash x^l : \tau$. Notice that

  $$\varphi^\bullet \circ g \circ (A'[x : \tau]) = (\varphi^\bullet \circ g \circ A')[x : (\varphi^\bullet \circ g(\tau))],$$

  and hence we have $(\varphi^\bullet \circ g \circ (A'[x : \tau]), x^l, \varphi^\bullet \circ g(\tau)) \in F(R)$.
- Rule (10). We have $e \equiv (\lambda x.e_1)^l$, and the last judgment of the derivation of $A \vdash e : \tau$ is of the form $A \vdash (\lambda x.e_1)^l : \tau$. We have $((\lambda x.e_1)^l, A) \in lamenv(\tau, D_E)$, so $\langle (\lambda x.e_1)^l, \varphi^\bullet \circ g \circ A \rangle \in \varphi^\bullet(Z_\tau) = \varphi^\bullet \circ g(\tau)$, and hence $(\varphi^\bullet \circ g \circ A, (\lambda x.e_1)^l, \varphi^\bullet \circ g(\tau)) \in F(R)$.
- Rule (11). We have $e \equiv (e_1\ e_2)^l$, and the last part of the derivation of $A \vdash e : \tau$ is of the form

  $$\frac{A \vdash e_1 : \sigma \to \tau \quad A \vdash e_2 : \sigma}{A \vdash (e_1 e_2)^l : \tau}$$

  From the two hypotheses of this, and from the definition of R, we have

  $$(\varphi^\bullet \circ g \circ A, e_1, \varphi^\bullet \circ g(\sigma \to \tau)) \in R$$
  $$(\varphi^\bullet \circ g \circ A, e_2, \varphi^\bullet \circ g(\sigma)) \in R.$$

  Suppose $\langle (\lambda x.e_3)^{l'}, \rho' \rangle \in \varphi^\bullet \circ g(\sigma \to \tau) = \varphi(Z_{\sigma \to \tau})$. From the definition of φ we have that $\rho' = \varphi^\bullet \circ g \circ A'$ for some A', and $((\lambda x.e_3)^{l'}, A') \in lamenv(\sigma \to \tau, D_E)$. From the definition of $lamenv$ we have that there exists a type δ such that $\delta \leqslant_1 (\sigma \to \tau)$ and $A' \vdash (\lambda x.e_3)^{l'} : \delta$ occurs in D_E, and the last rule used to

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

derive this judgment is Rule (10). We can write $\delta = \bigwedge_{k\in K}(\sigma_k \to \tau_k)$. The last part of the derivation of $A' \vdash (\lambda x.e_3)^{l'} : \delta$ is of the form

$$\frac{\forall k \in K : A'[x : \sigma_k] \vdash e_3 : \tau_k}{A' \vdash (\lambda x.e_3)^{l'} : \bigwedge_{k\in K}(\sigma_k \to \tau_k)}$$

From the hypotheses of this rule and definition of R we have, for all $k \in K$,

$$(\varphi^\bullet \circ g \circ (A'[x : \sigma_k]), e, \varphi^\bullet \circ g(\tau_k)) \in R.$$

Notice that, for all $k \in K$, $\varphi^\bullet \circ g \circ (A'[x : \sigma_k]) = (\varphi^\bullet \circ g \circ A')[x : \varphi^\bullet \circ g(\sigma_k)]$. We thus have, for all $k \in K$, $((\varphi^\bullet \circ g \circ A')[x : \varphi^\bullet \circ g(\sigma_k)], e, \varphi^\bullet \circ g(\tau_k)) \in R$. Define $C_J = \{\ \varphi^\bullet \circ g(\sigma_k) \mid k \in J\ \}$. It is now sufficient to show that there exists J such that $J \subseteq K$ and

$$\begin{aligned} \varphi^\bullet \circ g(\sigma) &\subseteq \bigcup_{k\in J}(\varphi^\bullet \circ g(\sigma_k)) \\ \bigcup_{k\in J}(\varphi^\bullet \circ g(\tau_k)) &\subseteq \varphi^\bullet \circ g(\tau). \end{aligned}$$

From Lemma 3.10 and $\delta \leqslant_1 \sigma \to \tau$ we have that there are two cases:

1. If $\delta = \sigma \to \tau$, then K is a singleton set, so we choose $J = K$ and the two properties are immediate.
2. If we have J such that $J \subseteq K$ and $\sigma \leqslant_1 \bigvee_{k\in J} \sigma_k$ and $\bigvee_{k\in J} \tau_k \leqslant_1 \tau$, then we use that particular J and then the two properties follow from Lemma 5.7.

- Rule (12). We have $e \equiv c^l$, and the derivation of $A \vdash e : \tau$ is of the form $A \vdash c^l : \mathsf{Int}$. Notice that $\varphi^\bullet \circ g(\mathsf{Int}) = \{\mathsf{Int}\}$, and hence we have $(\varphi^\bullet \circ g \circ A, c^l, \varphi^\bullet \circ g(\mathsf{Int})) \in F(R)$.
- Rule (13). We have $e \equiv (\mathsf{succ}\ e_1)^l$, and the last step of the derivation of $A \vdash e : \tau$ is of the form

$$\frac{A \vdash e_1 : \mathsf{Int}}{A \vdash (\mathsf{succ}\ e_1)^l : \mathsf{Int}}$$

From $A \vdash e_1 : \mathsf{Int}$ and the definition of R we have $(\varphi^\bullet \circ g \circ A, e_1, \varphi^\bullet \circ g(\mathsf{Int})) \in R$. Notice that $\varphi^\bullet \circ g(\mathsf{Int}) = \{\mathsf{Int}\}$, so $(\varphi^\bullet \circ g \circ A, (\mathsf{succ}\ e_1)^l, \{\mathsf{Int}\}) \in F(R)$.

- Rule (14). We have $e \equiv (\mathsf{if0}\ e_1\ e_2\ e_3)^l$, and the last part of the derivation of $A \vdash e : \tau$ is of the form

$$\frac{A \vdash e_1 : \mathsf{Int} \quad A \vdash e_2 : \tau \quad A \vdash e_3 : \tau}{A \vdash (\mathsf{if0}\ e_1\ e_2\ e_3)^l : \tau}$$

From the three hypothesis of this, and from the definition of R, we have

$$\begin{aligned} &(\varphi^\bullet \circ g \circ A, e_1, \varphi^\bullet \circ g(\mathsf{Int})) \in R, \\ &(\varphi^\bullet \circ g \circ A, e_2, \varphi^\bullet \circ g(\tau)) \in R, \\ &(\varphi^\bullet \circ g \circ A, e_3, \varphi^\bullet \circ g(\tau)) \in R, \end{aligned}$$

so

$$(\varphi^\bullet \circ g \circ A, (\mathsf{if0}\ e_1\ e_2\ e_3)^l, \varphi^\bullet \circ g(\tau)) \in F(R).$$

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

- Rule (15). The last part of the derivation of $A \vdash e : \tau$ is of the form

$$\frac{A \vdash e : \sigma}{A \vdash e : \tau} \quad (\sigma \leqslant_1 \tau)$$

From $A \vdash e : \sigma$ and the induction hypothesis we have

$$(\varphi^\bullet \circ g \circ A, e, \varphi^\bullet \circ g(\sigma)) \in F(R).$$

From $\sigma \leqslant_1 \tau$ and Lemma 5.7 we have $\varphi^\bullet \circ g(\sigma) \subseteq \varphi^\bullet \circ g(\tau)$, and from this and Lemma 4.1 we have

$$(\varphi^\bullet \circ g \circ A, e, \varphi^\bullet \circ g(\tau)) \in F(R).$$

□

In the proof of Theorem 5.8, the induction hypothesis is used only in the case of Rule (15). It is particularly important for the case of Rule (11) where $e \equiv (e_1\ e_2)^l$ that the induction hypothesis is *not* used to prove anything about $(\lambda x.e_3)^{l'}$, because $(\lambda x.e_3)^{l'}$ needs not be a subterm of e; it could occur anywhere.

Franklyn Turbak and Torben Amtoft have observed that the translation from flows to types and back to flows can lose precision. For example, consider the λ-term

$$E \quad = \quad ((\lambda x.10^1)^2\ ((\lambda y.30^3)^4\ 50^5)^6)^7$$

and the F-analysis

$$\begin{aligned} R \quad = \quad & \{(\emptyset[x : \{\mathsf{Int}\}], 10^1, \{\mathsf{Int}\}), \\ & (\emptyset, (\lambda x.10^1)^2, \{\langle (\lambda x.10^1)^2, \emptyset\rangle\}), \\ & (\emptyset[y : \{\mathsf{Int}\}], 30^3, \{\mathsf{Int}\}), \\ & (\emptyset, (\lambda y.30^3)^4, \{\langle (\lambda x.10^1)^2, \emptyset\rangle\}), \\ & (\emptyset, 50^5, \{\mathsf{Int}\}), \\ & (\emptyset, ((\lambda y.30^3)^4\ 50^5)^6, \{\mathsf{Int}\}), \\ & (\emptyset, E, \{\mathsf{Int}\})\}. \end{aligned}$$

The translation from flows to types maps R into the type derivation D:

$$\dfrac{\dfrac{\emptyset[x : \mathsf{Int}] \vdash 10^1 : \mathsf{Int}}{\emptyset \vdash (\lambda x.10^1)^2 : \mathsf{Int} \to \mathsf{Int}} \qquad \dfrac{\dfrac{\emptyset[y : \mathsf{Int}] \vdash 30^3 : \mathsf{Int}}{\emptyset \vdash (\lambda y.30^3)^4 : \mathsf{Int} \to \mathsf{Int}} \qquad \emptyset \vdash 50^5 : \mathsf{Int}}{\emptyset \vdash ((\lambda y.30^3)^4\ 50^5)^6 : \mathsf{Int}}}{\emptyset \vdash E : \mathsf{Int}}$$

Notice that $(\lambda x.10^1)^2, (\lambda y.30^3)^4$ both have type $\mathsf{Int} \to \mathsf{Int}$ in D. The translation from types to flows maps $\mathsf{Int} \to \mathsf{Int}$ to the set

$$s \quad = \quad \{\ \langle (\lambda x.10^1)^2, \emptyset\rangle, \langle (\lambda y.30^3)^4, \emptyset\rangle\ \}$$

and it maps D into an F-analysis that contains the judgments:

$$(\emptyset, (\lambda x.10^1)^2, s)$$
$$(\emptyset, (\lambda x.30^3)^4, s),$$

i.e. a less precise F-analysis than R.

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

6 Example

We will now illustrate how different flow analyses lead to different typings. We will give details of the flow analysis and typing of the λ-term

$$E = ((\lambda g.(\mathsf{if0}\ c\ (g^1\ g^2)^3\ (g^4\ (\lambda y.(y^5 0^6)^7)^8)^9)^{10})^{11}\ (\lambda f.(f^{12}\ (\lambda x.x^{13})^{14})^{15})^{16})^{17}.$$

This λ-term was suggested by Joe Wells, and it illustrates a key difference between 0-CFA and Schmidt's analysis. Like in the earlier example of the paper, we assume that the condition c of the if0-expression does not cause any run-time error. Written without the labels and with the notation $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ standing for $(\lambda x.e')e$, the λ-term E looks like:

$$\begin{array}{l}\mathsf{let}\ g = \lambda f.f(\lambda x.x)\\ \mathsf{in}\ \mathsf{if0}\ c\ (gg)\ (g(\lambda y.y0))\ .\end{array}$$

We present two flow analyses of E. One will be in the style of 0-CFA and one will be in the style of Schmidt, and we will show that they lead to different typings.

First, we do a 0-CFA-style analysis of the λ-term. Define $a_{\lambda g}, a_{\lambda y}, a_{\lambda f}, a_{\lambda x} \in \mathit{Closure}(E)$, $\rho \in \mathit{FlowEnv}(E)$ to be the unique solution to the next five equations, and define next $R_{0\text{-}CFA}$:

$$\begin{array}{rcl}
a_{\lambda g} & = & \langle(\lambda g.(\mathsf{if0}\ c\ (g^1\ g^2)^3\ (g^4\ (\lambda y.(y^5 0^6)^7)^8)^9)^{10})^{11}, \rho\rangle \\
a_{\lambda y} & = & \langle(\lambda y.(y^5 0^6)^7)^8, \rho\rangle \\
a_{\lambda f} & = & \langle(\lambda f.(f^{12}\ (\lambda x.x^{13})^{14})^{15})^{16}, \rho\rangle \\
a_{\lambda x} & = & \langle(\lambda x.x^{13})^{14}, \rho\rangle \\
\rho & = & \emptyset[g : \{a_{\lambda f}\}][y : \{a_{\lambda x}\}][f : \{a_{\lambda f}, a_{\lambda y}, a_{\lambda x}\}][x : \{a_{\lambda x}, \mathsf{Int}\}] \\
R_{0\text{-}CFA} & = & \{(\rho, g^1, \{\ a_{\lambda f}\ \}), \\
 & & \ (\rho, g^2, \{\ a_{\lambda f}\ \}), \\
 & & \ (\rho, (g^1\ g^2)^3, \{\ a_{\lambda x}, \mathsf{Int}\ \}), \\
 & & \ (\rho, g^4, \{\ a_{\lambda f}\ \}), \\
 & & \ (\rho, y^5, \{\ a_{\lambda x}\ \}), \\
 & & \ (\rho, 0^6, \{\ \mathsf{Int}\ \}), \\
 & & \ (\rho, (y^5\ 0^6)^7, \{\ a_{\lambda x}, \mathsf{Int}\ \}), \\
 & & \ (\rho, (\lambda y.(y^5 0^6)^7)^8, \{\ a_{\lambda y}\ \}), \\
 & & \ (\rho, (g^4\ (\lambda y.(y^5 0^6)^7)^8)^9, \{\ a_{\lambda x}, \mathsf{Int}\ \}), \\
 & & \ (\rho, (\mathsf{if0}\ c\ (g^1\ g^2)^3\ (g^4\ (\lambda y.(y^5 0^6)^7)^8)^9)^{10}, \{\ a_{\lambda x}, \mathsf{Int}\ \}), \\
 & & \ (\rho, (\lambda g.(\mathsf{if0}\ c\ (g^1\ g^2)^3\ (g^4\ (\lambda y.(y^5 0^6)^7)^8)^9)^{10})^{11}, \{\ a_{\lambda g}\ \}), \\
 & & \ (\rho, f^{12}, \{\ a_{\lambda f}, a_{\lambda y}, a_{\lambda x}\ \}) \\
 & & \ (\rho, x^{13}, \{\ a_{\lambda x}, \mathsf{Int}\ \}), \\
 & & \ (\rho, (\lambda x.x^{13})^{14}, \{\ a_{\lambda x}\ \}), \\
 & & \ (\rho, (f^{12}\ (\lambda x.x^{13})^{14})^{15}, \{\ a_{\lambda x}, \mathsf{Int}\ \}), \\
 & & \ (\rho, (\lambda f.(f^{12}\ (\lambda x.x^{13})^{14})^{15})^{16}, \{\ a_{\lambda f}\ \}), \\
 & & \ (\rho, E, \{\ a_{\lambda x}, \mathsf{Int}\ \}), \\
 & & \}
\end{array}$$

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

Notice that we have $(\rho,(y^5\ 0^6)^7,\{\ a_{\lambda x},\ \mathsf{Int}\ \})$ rather than $(\rho,(y^5\ 0^6)^7,\{\ \mathsf{Int}\ \})$. This is because y can evaluate to $(\lambda x.x^{13})^{14}$, and $(\lambda x.x^{13})^{14}$ is applied to both $(\lambda x.x^{13})^{14}$ and an integer, so the flow information for x is $\{a_{\lambda x},\mathsf{Int}\}$. Similarly, we have $(\rho,(g^1\ g^2)^3,\{a_{\lambda x},\mathsf{Int}\})$ rather than $(\rho,(g^1\ g^2)^3,\{a_{\lambda x}\})$. The reader is invited to check that $R_{0\text{-}CFA} \subseteq F(R_{0\text{-}CFA})$, $R_{0\text{-}CFA}$ is safe, and $R_{0\text{-}CFA}$ analyzes all its closure bodies.

Notice that

$$collect(R_{0\text{-}CFA}) \quad = \quad \{\ a_{\lambda g},\ a_{\lambda f},\ a_{\lambda y},\ a_{\lambda x},\ \mathsf{Int}\ \}.$$

Notice also that

$$\begin{aligned}
argres(a_{\lambda g}, R_{0\text{-}CFA}) &= \{\ (\{a_{\lambda f}\},\{a_{\lambda x},\mathsf{Int}\})\ \}\\
argres(a_{\lambda y}, R_{0\text{-}CFA}) &= \{\ (\{a_{\lambda x}\},\{a_{\lambda x},\mathsf{Int}\})\ \}\\
argres(a_{\lambda f}, R_{0\text{-}CFA}) &= \{\ (\{a_{\lambda f},a_{\lambda y},a_{\lambda x}\},\{a_{\lambda x},\mathsf{Int}\})\ \}\\
argres(a_{\lambda x}, R_{0\text{-}CFA}) &= \{\ (\{a_{\lambda x},\mathsf{Int}\},\{a_{\lambda x},\mathsf{Int}\})\ \}
\end{aligned}$$

The equation system $TypeEqSys(R_{0\text{-}CFA})$ contains the following four equations:

$$\begin{aligned}
W_{a_{\lambda g}} &= W_{a_{\lambda f}} \to (W_{a_{\lambda x}} \vee \mathsf{Int})\\
W_{a_{\lambda y}} &= W_{a_{\lambda x}} \to (W_{a_{\lambda x}} \vee \mathsf{Int})\\
W_{a_{\lambda f}} &= (W_{a_{\lambda f}} \vee W_{a_{\lambda y}} \vee W_{a_{\lambda x}}) \to (W_{a_{\lambda x}} \vee \mathsf{Int})\\
W_{a_{\lambda x}} &= (W_{a_{\lambda x}} \vee \mathsf{Int}) \to (W_{a_{\lambda x}} \vee \mathsf{Int})
\end{aligned}$$

Here is a solution ψ of $TypeEqSys(R_{0\text{-}CFA})$:

$$\begin{aligned}
\psi(W_{a_{\lambda g}}) &= \tau' \to \sigma\\
\psi(W_{a_{\lambda y}}) &= (\sigma \to \sigma) \to \sigma\\
\psi(W_{a_{\lambda f}}) &= \tau'\\
\psi(W_{a_{\lambda x}}) &= \sigma \to \sigma
\end{aligned}$$

where

$$\begin{aligned}
\tau &= \mu\alpha.((\alpha \vee \mathsf{Int}) \to (\alpha \vee \mathsf{Int}))\\
\sigma &= \tau \vee \mathsf{Int}\\
\tau' &= \mu\alpha.((\alpha \vee ((\sigma \to \sigma) \to \sigma) \vee (\sigma \to \sigma)) \to \sigma).
\end{aligned}$$

Notice that $\sigma \to \sigma = \tau \leqslant \sigma$. To see that τ' is a type for $(\lambda f.(f^{12}\ (\lambda x.x^{13})^{14})^{15})^{16}$, notice that

$$\tau' = [\tau' \vee ((\sigma \to \sigma) \to \sigma) \vee (\sigma \to \sigma)] \to \sigma$$

and from Lemma (3.10) we have

$$\tau' \vee ((\sigma \to \sigma) \to \sigma) \vee (\sigma \to \sigma)\ \leqslant_1\ (\sigma \to \sigma) \to \sigma.$$

From this we conclude that f^{12} has type $(\sigma \to \sigma) \to \sigma$, and moreover $(\lambda x.x^{13})^{14}$ has type $\sigma \to \sigma$, so $(f^{12}\ (\lambda x.x^{13})^{14})^{15}$ has type σ, as required.

Next, we will do a Schmidt-style analysis of the λ-term. It turns out that for E, the Schmidt-style analysis gives the same result as an Agesen-style analysis, since

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

the argument sets of all the argument-result pairs are singletons, see below. Define

$$\begin{aligned} a^1_{\lambda g}, a^1_{\lambda y}, a^1_{\lambda f}, a^1_{\lambda x}, a^2_{\lambda x}, a^3_{\lambda x} &\in \mathit{Closure}(E), \\ \rho_{21}, \rho_{22}, \rho_{23}, \rho_{24}, \rho_{2h}, \rho_{26}, \rho_{27}, \rho_{28} &\in \mathit{FlowEnv}(E), \\ R_{Schmidt} &\in \mathit{FlowJudgmentSet}(E) \end{aligned}$$

as follows:

$$\begin{aligned} a^1_{\lambda g} &= \langle (\lambda g.(\mathsf{if0}\ c\ (g^1\ g^2)^3\ (g^4\ (\lambda y.(y^5 \mathsf{0}^6)^7)^8)^9)^{10})^{11}, \emptyset \rangle \\ a^1_{\lambda y} &= \langle (\lambda y.(y^5 \mathsf{0}^6)^7)^8, \rho_{21} \rangle \\ a^1_{\lambda f} &= \langle (\lambda f.(f^{12}\ (\lambda x.x^{13})^{14})^{15})^{16}, \emptyset \rangle \\ a^1_{\lambda x} &= \langle (\lambda x.x^{13})^{14}, \rho_{22} \rangle \\ a^2_{\lambda x} &= \langle (\lambda x.x^{13})^{14}, \rho_{23} \rangle \\ a^3_{\lambda x} &= \langle (\lambda x.x^{13})^{14}, \rho_{25} \rangle \\ \rho_{21} &= \{\ g : \{a^1_{\lambda f}\}\ \} \\ \rho_{22} &= \{\ f : \{a^1_{\lambda f}\}\ \} \\ \rho_{23} &= \{\ f : \{a^1_{\lambda y}\}\ \} \\ \rho_{24} &= \rho_{21}[y : \{a^2_{\lambda x}\}] \\ \rho_{25} &= \{\ f : \{a^1_{\lambda x}\}\ \} \\ \rho_{26} &= \rho_{22}[x : \{a^3_{\lambda x}\}] \\ \rho_{27} &= \rho_{23}[x : \{\mathsf{Int}\}] \\ \rho_{28} &= \rho_{25}[x : \emptyset] \\ R_{Schmidt} &= \{\ (\rho_{21}, g^1, \{\ a^1_{\lambda f}\ \}), \\ &\quad (\rho_{21}, g^2, \{\ a^1_{\lambda f}\ \}), \\ &\quad (\rho_{21}, (g^1\ g^2)^3, \{\ a^3_{\lambda x}\ \}), \\ &\quad (\rho_{21}, g^4, \{\ a^1_{\lambda f}\ \}), \\ &\quad (\rho_{24}, y^5, \{\ a^2_{\lambda x}\ \}), \\ &\quad (\rho_{24}, \mathsf{0}^6, \{\ \mathsf{Int}\ \}), \\ &\quad (\rho_{24}, (y^5\ \mathsf{0}^6)^7, \{\ \mathsf{Int}\ \}), \\ &\quad (\rho_{21}, (\lambda y.(y^5 \mathsf{0}^6)^7)^8, \{\ a^1_{\lambda y}\ \}), \\ &\quad (\rho_{21}, (g^4\ (\lambda y.(y^5 \mathsf{0}^6)^7)^8)^9, \{\ \mathsf{Int}\ \}), \\ &\quad (\rho_{21}, (\mathsf{if0}\ c\ (g^1\ g^2)^3\ (g^4\ (\lambda y.(y^5 \mathsf{0}^6)^7)^8)^9)^{10}, \{\ a^3_{\lambda x},\ \mathsf{Int}\ \}), \\ &\quad (\emptyset, (\lambda g.(\mathsf{if0}\ c\ (g^1\ g^2)^3\ (g^4\ (\lambda y.(y^5 \mathsf{0}^6)^7)^8)^9)^{10})^{11}, \{\ a^1_{\lambda g}\ \}), \\ &\quad (\rho_{22}, f^{12}, \{\ a^1_{\lambda f}\ \}) \\ &\quad (\rho_{23}, f^{12}, \{\ a^1_{\lambda y}\ \}) \\ &\quad (\rho_{25}, f^{12}, \{\ a^1_{\lambda x}\ \}) \\ &\quad (\rho_{26}, x^{13}, \{\ a^3_{\lambda x}\ \}), \\ &\quad (\rho_{27}, x^{13}, \{\ \mathsf{Int}\ \}), \\ &\quad (\rho_{28}, x^{13}, \emptyset), \end{aligned}$$

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

$$
\begin{array}{l}
(\rho_{22}, (\lambda x.x^{13})^{14}, \{\ a^1_{\lambda x}\ \}),\\
(\rho_{23}, (\lambda x.x^{13})^{14}, \{\ a^2_{\lambda x}\ \}),\\
(\rho_{25}, (\lambda x.x^{13})^{14}, \{\ a^3_{\lambda x}\ \}),\\
(\rho_{22}, (f^{12}\ (\lambda x.x^{13})^{14})^{15}, \{\ a^3_{\lambda x}\ \}),\\
(\rho_{23}, (f^{12}\ (\lambda x.x^{13})^{14})^{15}, \{\ \mathsf{Int}\ \}),\\
(\rho_{25}, (f^{12}\ (\lambda x.x^{13})^{14})^{15}, \{\ a^3_{\lambda x}\ \}),\\
(\emptyset, (\lambda f.(f^{12}\ (\lambda x.x^{13})^{14})^{15})^{16}, \{\ a^1_{\lambda f}\ \}),\\
(\emptyset, E, \{\ a^3_{\lambda x},\ \mathsf{Int}\ \}),\\
\}
\end{array}
$$

Notice that $(\rho_{28}, x^{13}, \emptyset) \in R_{Schmidt}$, reflecting that E can evaluate to a copy of $(\lambda x.x^{13})^{14}$ that is never applied. The judgment $(\rho_{28}, x^{13}, \emptyset)$ helps ensure that $R_{Schmidt}$ analyzes all its closure bodies. The reader is invited to check that $R_{Schmidt} \subseteq F(R_{Schmidt})$, $R_{Schmidt}$ is safe, and $R_{Schmidt}$ analyzes all its closure bodies.

Notice that

$$collect(R_{Schmidt}) \quad = \quad \{\ a^1_{\lambda g},\ a^1_{\lambda f},\ a^1_{\lambda y},\ a^1_{\lambda x},\ a^2_{\lambda x},\ a^3_{\lambda x},\ \mathsf{Int}\ \}.$$

Notice also that

$$
\begin{array}{lcl}
argres(a^1_{\lambda g}, R_{Schmidt}) & = & \{\ (\{a^1_{\lambda f}\}, \{a^3_{\lambda x}, \mathsf{Int}\})\ \}\\
argres(a^1_{\lambda y}, R_{Schmidt}) & = & \{\ (\{a^2_{\lambda x}\}, \{\mathsf{Int}\})\ \}\\
argres(a^1_{\lambda f}, R_{Schmidt}) & = & \{\ (\{a^1_{\lambda f}\}, \{a^3_{\lambda x}\}),\ (\{a^1_{\lambda y}\}, \{\mathsf{Int}\}),\ (\{a^1_{\lambda x}\}, \{a^3_{\lambda x}\})\ \}\\
argres(a^1_{\lambda x}, R_{Schmidt}) & = & \{\ (\{a^3_{\lambda x}\}, \{a^3_{\lambda x}\})\ \}\\
argres(a^2_{\lambda x}, R_{Schmidt}) & = & \{\ (\{\mathsf{Int}\}, \{\mathsf{Int}\})\ \}\\
argres(a^3_{\lambda x}, R_{Schmidt}) & = & \{\ (\emptyset, \emptyset)\ \}
\end{array}
$$

The equation system $TypeEqSys(R_{Schmidt})$ contains the following six equations:

$$
\begin{array}{lcl}
W_{a^1_{\lambda g}} & = & W_{a^1_{\lambda f}} \to (W_{a^3_{\lambda x}} \vee \mathsf{Int})\\
W_{a^1_{\lambda y}} & = & W_{a^2_{\lambda x}} \to \mathsf{Int}\\
W_{a^1_{\lambda f}} & = & (W_{a^1_{\lambda f}} \to W_{a^3_{\lambda x}}) \wedge (W_{a^1_{\lambda y}} \to \mathsf{Int}) \wedge (W_{a^1_{\lambda x}} \to W_{a^3_{\lambda x}})\\
W_{a^1_{\lambda x}} & = & W_{a^3_{\lambda x}} \to W_{a^3_{\lambda x}}\\
W_{a^2_{\lambda x}} & = & \mathsf{Int} \to \mathsf{Int}\\
W_{a^3_{\lambda x}} & = & \bot \to \bot
\end{array}
$$

Here is a solution ψ of $TypeEqSys(R_{Schmidt})$:

$$
\begin{array}{lcl}
\psi(W_{a^1_{\lambda g}}) & = & \tau'' \to ((\bot \to \bot) \vee \mathsf{Int})\\
\psi(W_{a^1_{\lambda y}}) & = & (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}\\
\psi(W_{a^1_{\lambda f}}) & = & \tau''\\
\psi(W_{a^1_{\lambda x}}) & = & (\bot \to \bot) \to (\bot \to \bot)\\
\psi(W_{a^2_{\lambda x}}) & = & \mathsf{Int} \to \mathsf{Int}\\
\psi(W_{a^3_{\lambda x}}) & = & \bot \to \bot
\end{array}
$$

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

where

$$\begin{aligned}\tau'' \quad = \quad & \mu\alpha.((\alpha \to (\bot \to \bot)) \wedge \\ & (((\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Int}) \wedge \\ & (((\bot \to \bot) \to (\bot \to \bot)) \to (\bot \to \bot))).\end{aligned}$$

The typing produced via the Schmidt-style analysis is more precise. For example, it gives $\lambda y.y\mathsf{0}$ the type $(\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$, where 0-CFA leads to the type $(\sigma \to \sigma) \to \sigma$, where $\sigma = \tau \vee \mathsf{Int}$, and $\tau = \mu\alpha.((\alpha \vee \mathsf{Int}) \to (\alpha \vee \mathsf{Int}))$.

Palsberg and O'Keefe (1995) presented a mapping from 0-CFA flow information to the types of Amadio and Cardelli (1993). The type system of Amadio and Cardelli (1993) involves subtyping and recursive types but not intersection and union types. We will now compare the mapping of this paper with the mapping of Palsberg and O'Keefe. We do this by showing the result of mapping the 0-CFA flow information for the λ-term E studied in this section to the types of Amadio and Cardelli using the mapping of Palsberg and O'Keefe. For our purposes here, we will not need the recursive types so we only recall the non-recursive fragment of the Amadio/Cardelli type system. Types are defined by the following grammar:

$$t, u \quad ::= \quad t \to t \mid \mathsf{Int} \mid \bot \mid \top.$$

The type rules are the same as in section 3, with the straightforward modification that Rule (10), that is, the rule for λ-abstraction, introduces just a single function type rather than an intersection of function types. Subtyping is defined by the rules:

$$\frac{t' \leqslant t \qquad u \leqslant u'}{t \to u \leqslant t' \to u'} \qquad \bot \leqslant t \qquad t \leqslant \top \qquad \mathsf{Int} \leqslant \mathsf{Int}.$$

It is straightforward to show that $\leqslant$ is reflexive and transitive.

The mapping of Palsberg and O'Keefe maps the 0-CFA flow information for E to a type derivation for E which we sketch here:

$$\begin{aligned}
&A = \emptyset[g : ((\top \to \top) \to \top) \to \top, \\
&\qquad y : \top \to \top, \\
&\qquad f : (\top \to \top) \to \top, \\
&\qquad x : \top] \\
&A \vdash (g^1\ g^2)^3 : \top \\
&A \vdash \mathsf{0}^6 : \top \\
&A \vdash (\lambda y.(y^5 \mathsf{0}^6)^7)^8 : (\top \to \top) \to \top \\
&A \vdash (\lambda g.(\mathsf{if0}\ c\ (g^1\ g^2)^3\ (g^4\ (\lambda y.(y^5 \mathsf{0}^6)^7)^8)^9)^{10})^{11} : \\
&\qquad\qquad (((\top \to \top) \to \top) \to \top) \to \top \\
&A \vdash (\lambda x.x^{13})^{14} : \top \to \top \\
&A \vdash (\lambda f.(f^{12}\ (\lambda x.x^{13})^{14})^{15})^{16} : (((\top \to \top) \to \top) \to \top \\
&A \vdash E : \top.
\end{aligned}$$

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

Subtyping is needed twice in that type derivation: 1) to show $A \vdash (g^1\ g^2)^3 : \top$ we need

$$((\top \to \top) \to \top) \to \top \ \leqslant \ (\top \to \top) \to \top,$$

and 2) to show $A \vdash 0^6 : \top$ we need $\mathsf{Int} \leqslant \top$.

Compared with the previous 0-CFA-based typing, the biggest difference is the type for $(\lambda f.(f^{12}\ (\lambda x.x^{13})^{14})^{15})^{16}$. The type $\top$ plays the role of $\sigma = \tau \vee \mathsf{Int}$.

It is straightforward to show that E cannot be typed in the fragment of the Amadio/Cardelli type system which excludes the rule for comparing function types. This was noticed by Joe Wells.

7 A flow-type system

In a flow-type system, types and flow information are combined. Such flow-type systems have been studied by numerous authors (Tang and Jouvelot, 1994; Heintze, 1995; Banerjee, 1997; Wells *et al.*, 1997; Turbak *et al.*, 1997; Dimock *et al.*, 1997). A flow-type system is useful as an interface between a flow-analysis algorithm and a program optimizer. We will use our equivalence theorem to guide the design of a new flow-type system. Our flow-type system is based on three design decisions:

1. We want the set of typable terms to be the same as the set of terms typable in $\mathscr{T}_{\leqslant_1}$. We achieve that by annotating the types from $\mathscr{T}_{\leqslant_1}$ with flow information.
2. We want to annotate the types in a way which is suggested by the mapping from flows to types in section 5.2. The key property of that mapping is that a closure is mapped to an intersection type. Thus, we choose to annotate the intersection types.
3. Following Wells *et al.* (1997), we want to annotate the intersection types with sets of labels, not sets of closures.

We use π to range over finite sets of labels. Flow types are of one of the forms:

$$\bigvee_{i\in I}\left(\bigwedge_{k\in K}(\sigma_{ik} \to \sigma'_{ik})\right)^{\pi_i}$$
$$\left(\bigvee_{i\in I}\left(\bigwedge_{k\in K}(\sigma_{ik} \to \sigma'_{ik})\right)^{\pi_i}\right) \vee \mathsf{Int}.$$

A precise definition of the set of types and of type equality can be given like in section 3, we omit the details. We use *FlowType* to denote the set of flow types. We use δ, σ, τ to range over flow types. We use *FlowIntersectionType* to the denote the set of flow types of the form $(\bigwedge_{k\in K}(\sigma_k \to \sigma'_k))^{\pi}$. We use Q to range over *FlowIntersectionType*. We use T to range over phrases of the form $\bigwedge_{k\in K}(\sigma_{ik} \to \sigma'_{ik})$. We use u to range over type expressions of the forms Int and Q.

Definition 7.1

(Acceptable Flow-Type Orderings) We say that an ordering $\leqslant$ on flow types is *acceptable* if and only if $\leqslant$ satisfies the five conditions:

1. $\leqslant$ is reflexive,

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

2. $\leqslant$ is transitive,
3. if $(\bigwedge_{k\in K}(\sigma_k \to \sigma'_k))^\pi \leqslant (\tau_1 \to \tau_2)^{\pi'}$, and $u \leqslant \tau_1$, then there exists $k_0 \in K$ such that $u \leqslant \sigma_{k_0}$ and $\sigma'_{k_0} \leqslant \tau_2$,
4. $(\bigwedge_{k\in K}(\sigma_k \to \tau_k))^\pi \not\leqslant \mathsf{Int}$, and
5. $\mathsf{Int} \not\leqslant (\sigma \to \tau)^\pi$.

As in section 3.4, the type system is parameterized by a type ordering. Given a type ordering $\leqslant$, we will inductively define the set $\mathscr{T}^F_{\leqslant}$ of valid type judgments. Rules are (9)–(15), except that the rules for abstraction and application are modified to look as follows:

$$\frac{\forall k \in K : A[x : \sigma_k] \vdash e : \tau_k}{A \vdash (\lambda x.e)^l : (\bigwedge_{k\in K}(\sigma_k \to \tau_k))^\pi} \quad (l \in \pi) \tag{24}$$

$$\frac{A \vdash e_1 : (\sigma \to \tau)^\pi \quad A \vdash e_2 : \sigma}{A \vdash (e_1 e_2)^l : \tau} \tag{25}$$

To obtain a type preservation result, we can repeat the proofs of Theorem 3.6 and the associated lemmas, with small modifications, and obtain the following result. We omit the proof.

Theorem 7.2
(Flow-Type Preservation) For an acceptable flow-type ordering $\leqslant$, if $\mathscr{T}^F_{\leqslant} \triangleright A \vdash e : \tau$ and $e \to_V e'$, then $\mathscr{T}^F_{\leqslant} \triangleright A \vdash e' : \tau$.

Define

$$\begin{aligned} LabSet &: FlowType \to \mathscr{P}(Lab) \\ LabSet(\mathsf{Int}) &= \emptyset \\ LabSet(\bot) &= \emptyset \\ LabSet(\tau \vee \tau') &= LabSet(\tau) \cup LabSet(\tau') \\ LabSet(T^\pi) &= \pi \end{aligned}$$

We say that an ordering $\leqslant$ on flow types *respects flow* if and only if

$$\text{if } \sigma \leqslant \tau, \text{ then } LabSet(\sigma) \subseteq LabSet(\tau).$$

Theorem 7.3
(Flow Soundness) For an acceptable flow-type ordering $\leqslant$ which respects flow, if $\mathscr{T}^F_{\leqslant} \triangleright A \vdash e : \tau$ and $e \to^*_V (\lambda x.e')^l$, then $l \in LabSet(\tau)$.

Proof
From Theorem 7.2 and an induction argument we have $\mathscr{T}^F_{\leqslant} \triangleright A \vdash (\lambda x.e')^l : \tau$. There are now two cases depending on which flow-type rule was the last one used in the derivation of $A \vdash (\lambda x.e')^l : \tau$.

- Rule (24). We have $\tau = (\bigwedge_{k\in K}(\sigma_k \to \tau_k))^\pi$, where $l \in \pi = LabSet(\tau)$.

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

- Rule (15). The last part of the derivation of $A \vdash (\lambda x.e')^l : \tau$ is of the form

$$\frac{A \vdash e : \sigma}{A \vdash e : \tau} \quad (\sigma \leqslant \tau)$$

where $\sigma = (\bigwedge_{k \in K}(\sigma_k \to \tau_k))^\pi$, and $l \in \pi = LabSet(\sigma) \subseteq LabSet(\tau)$.

□

We now define an acceptable flow-type ordering $\leqslant_2$ which respects flow. We write $\sigma \leqslant_2 \tau$ if and only if we can derive $\sigma \leqslant_2 \tau$ using the following rules.

$$\frac{\sigma \leqslant_2 \delta \qquad \delta \leqslant_2 \tau}{\sigma \leqslant_2 \tau}$$

$$\sigma \leqslant_2 \sigma \vee \tau'$$

$$\frac{\forall i \in I : \sigma_i \leqslant_2 (\tau_1 \to \tau_2)^\pi}{\bigvee_{i \in I} \sigma_i \leqslant_2 (\tau_1 \to \tau_2)^\pi}$$

$$\frac{\tau_1 \leqslant_2 \sigma_1 \qquad \sigma_2 \leqslant_2 \tau_2}{(\sigma_1 \to \sigma_2)^{\pi'} \leqslant_2 (\tau_1 \to \tau_2)^\pi} \quad (\pi' \subseteq \pi)$$

$$\frac{(\bigwedge_{k \in K}(\sigma_k \to \sigma'_k))^{\pi'} \leqslant_2 (\tau_1 \to \tau_2)^\pi}{(\bigwedge_{k \in K'}(\sigma_k \to \sigma'_k))^{\pi'} \leqslant_2 (\tau_1 \to \tau_2)^\pi} \quad (K \subseteq K')$$

$$\left(\bigwedge_{k \in K}(\sigma_k \to \sigma'_k)\right)^\pi \leqslant_2 \left(\left(\bigvee_{k \in K} \sigma_k\right) \to \left(\bigvee_{k \in K} \sigma'_k\right)\right)^\pi$$

Theorem 7.4
The relation $\leqslant_2$ is an acceptable flow-type ordering which respects flow.

We omit the proof; it is similar to the proof of Theorem 3.12 with a straightforward extension to show that $\leqslant_2$ respects flow.

It is straightforward to show that a program is typable in $\mathscr{T}_{\leqslant_1}$ if and only if it is typable in $\mathscr{T}^F_{\leqslant_2}$. From this observation and Corollary 3.9, we get that a program typable in $\mathscr{T}^F_{\leqslant_2}$ cannot go wrong.

We invite the reader to construct a flow-type derivation for the example program in section 6.

This completes our development of the flow-type system. Let us now compare it with the one of Wells *et al.* (1997). Notable differences include:

- In Wells *et al.* (1997), there is a general $\wedge$-introduction rule. In our system, $\wedge$ can only be introduced at the point of typing a λ-abstraction.
- In Wells *et al.* (1997), individual function types are annotated. In our system, intersection types (possibly consisting of just one function type) are annotated.
- In Wells *et al.* (1997), the types are annotated with two labels sets, one for abstraction labels and one for call-site labels. In our system, the types are annotated with just one label set, although it is straightforward to extend our system to annotate in the style of Wells *et al.* (1997).

https://doi.org/10.1017/S095679680100394X Published online by Cambridge University Press

- In Wells *et al.* (1997), the calculus is explicitly typed. Our calculus is implicitly typed. It should be possible to construct an explicitly-typed version of our calculus, using ideas from Wells *et al.* (1997).
- In Wells *et al.* (1997), subtyping for function types allows adding abstraction labels and removing call-site labels. Our notion of subtyping also allows adding abstraction labels and, in addition, it allows 'deep subtyping', i.e. changes to the argument and the result type.
- In Wells *et al.* (1997), intersection and union types do not enjoy any algebraic laws like they do in our calculus. The design choice in Wells *et al.* (1997) has advantages in the setting of program optimization; see Wells *et al.* (1997) for details.

In spite of these differences, the calculi in Wells *et al.* (1997) and this paper are quite similar. Derived systematically via our equivalence theorem, our flow-type system should be a good interface to the family of polyvariant analyses that we study.

Acknowledgements

We thank Allyn Dimock, Assaf Kfoury, Robert Muller, Franklyn Turbak and Joe Wells for many discussions on intersection and union types. We thank Mitchell Wand for many discussions on polyvariant flow analysis. We thank Torben Amtoft and the reviewers for Journal of Functional Programming and POPL 1998 for detailed and insightful comments on drafts of the paper. We thank Joe Wells for noticing that the analysis in the POPL 1998 version of the paper was not sufficiently general to cover 0-CFA. Finally, we thank Philip Wadler for encouragement during the revision of the paper. Palsberg is supported by a National Science Foundation Faculty Early Career Development Award, CCR–9734265.

References

Agesen, O. (1995a) The Cartesian product algorithm. *Proc. ECOOP'95, 7th European Conference on Object-Oriented Programming: Lecture Notes in Computer Science 952*, pp. 2–26. Springer-Verlag.

Agesen, O. (1995b) *Concrete type inference: Delivering object-oriented applications*. PhD thesis, Stanford University.

Agesen, O. and Ungar, D. (1994) Sifting out the gold: Delivering compact applications from an exploratory object-oriented environment. *Proc. OOPSLA'94, ACM SIGPLAN 9th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 355–370.

Aiken, A., Wimmers, E. L. and Lakshman, T. K. (1994) Soft typing with conditional types. *Proc. POPL'94, 21st Annual Symposium on Principles of Programming Languages*, pp. 163–173.

Aiken, A. and Wimmers, E. (1993) Type inclusion constraints and type inference. *Proc. Conference on Functional Programming Languages and Computer Architecture*, pp. 31–41.

Amadio, R. M. and Cardelli, L. (1993) Subtyping recursive types. *ACM Trans. Programming Languages and Syst.*, **15**(4), 575–631.

Amtoft, T. (1993) Minimal thunkification. *Proc. WSA'93, 3rd International Workshop on Static Analysis: Lecture Notes in Computer Science 724*, pp. 218–229. Springer-Verlag.

Appel, A. W. (1992) *Compiling with Continuations*. Cambridge University Press.

Banerjee, A. (1997) A modular, polyvariant and type-based closure analysis. *Proc. ACM International Conference on Functional Programming*.

Barbanera, F., Dezani-Ciancaglini, M. and De'Liguoro, U. (1995) Intersection and union types: Syntax and semantics. *Information & Computation*, **119**(2), 202–230.

Barendregt, H. P. (1981) *The Lambda Calculus: Its syntax and semantics*. North-Holland.

Bondorf, A. (1991) Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, **17**(1–3), 3–34.

Consel, C. (1993) A tour of Schism: A partial evaluation system for higher-order applicative languages. *Proc. PEPM'93, 2nd ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pp. 145–154.

Coppo, M., Dezani-Ciancaglini, M. and Venneri, B. (1980) Principal type schemes and lambda-calculus semantics. In: Seldin, J. and Hindley, J. (editors), *To H. B. Curry: Essays on combinatory logic, lambda calculus and formalism*, pp. 535–560. Academic Press.

Coppo, M. and Giannini, P. (1992) A complete type inference algorithm for simple intersection types. *Proc. CAAP'92: Lecture Notes in Computer Science 581*, pp. 102–123. Springer-Verlag.

Courcelle, B. (1983) Fundamental properties of infinite trees. *Theor. Comput. Sci.*, **25**(1), 95–169.

Cousot, P. (1997) Types as abstract interpretations. *Proc. POPL'97, 24th Annual SIGPLAN–SIGACT Symposium on Principles of Programming languages*, pp. 316–331.

Despeyroux, J. (1986) Proof of translation in natural semantics. *LICS'86, 1st Symposium on Logic in Computer Science*, pp. 193–205.

Dimock, A., Muller, R., Turbak, F. and Wells, J. B. (1997) Strongly typed flow-directed representation transformations. *Proc. ICFP'97, International Conference on Functional Programming*, pp. 11–24.

Eifrig, J., Smith, S. and Trifonov, V. (1995a) Sound polymorphic type inference for objects. *Proc. OOPSLA'95, ACM SIGPLAN 10th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 169–184.

Eifrig, J., Smith, S. and Trifonov, V. (1995b) Type inference for recursively constrained types and its application to OOP. *Proc. Mathematical Foundations of Programming Semantics*. Elsevier Electronic Notes in Theoretical Computer Science, vol. 1.

Emami, M., Ghiya, R. and Hendren, L. J. (1994) Context-sensitive interprocedural points-to analysis in the presence of function pointers. *Proc. ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pp. 242–256.

Faxén, K.-F. (1995) Optimizing lazy functional programs using flow inference. *Proc. SAS'95, International Static Analysis Symposium: Lecture Notes in Computer Science 983*, pp. 242–256. Springer-Verlag.

Graver, J. O. and Johnson, R. E. (1990) A type system for Smalltalk. *17th Symposium on Principles of Programming Languages*, pp. 136–150.

Grove, D., DeFouw, G., Dean, J. and Chambers, C. (1997) Call graph construction in object-oriented languages. *Proc. OOPSLA'97, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*.

Heintze, N. (1995) Control-flow analysis and type systems. *Proc. SAS'95, International Static Analysis Symposium: Lecture Notes in Computer Science 983*. Springer-Verlag.

Heintze, N. and McAllester, D. (1997a) Linear-time subtransitive control flow analysis. *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.261–272.

Heintze, N. and McAllester, D. (1997b) On the complexity of set based analysis. *Proc. ACM International Conference on Functional Programming*.

Henglein, F. and Mossin, C. (1994) Polymorphic binding time analysis. *Proc. ESOP'94, European Symposium on Programming: Lecture Notes in Computer Science 788*, pp. 287–301. Springer-Verlag.

Hindley, J. R. (1991) Types with intersection: An introduction. *Formal Aspects of Computing*, **4**, 470–486.

Jagannathan, S. and Weeks, S. (1995) A unified treatment of flow analysis in higher-order languages. *Proc. POPL'95, 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pp. 393–407.

Jagannathan, S. and Wright, A. (1995) Effective flow analysis for avoiding run-time checks. *Proc. SAS'95, International Static Analysis Symposium: Lecture Notes in Computer Science 983*. Springer-Verlag.

Jagannathan, S., Wright, A. and Weeks, S. (1997) Type-directed flow analysis for typed intermediate languages. *Proc. SAS'95, International Static Analysis Symposium: Lecture Notes in Computer Science 983*. Springer-Verlag.

Jim, T. (1996a) *Principal typings and type inference*. PhD thesis, MIT.

Jim, T. (1996b) What are principal typings and what are they good for? *Proc. POPL'96, 23nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pp. 42–53.

Jones, N. and Muchnick, S. (1982) A flexible approach to interprocedural data flow analysis of programs with recursive data structures. *Ninth Symposium on Principles of Programming Languages*, pp. 66–74.

Kahn, G. (1987) Natural semantics. *Proc. STACS'87: Lecture Notes in Computer Science 247*, pp. 22–39. Springer-Verlag.

Kozen, D., Palsberg, J. and Schwartzbach, M. I. (1995) Efficient recursive subtyping. *Mathematical Structures in Comput. Sci.*, **5**(1), 113–125. (Preliminary version in *Proc. POPL'93, 20th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pp. 419–428. Charleston, SC, January 1993.)

Kuo, T.-M. and Mishra, P. (1989) Strictness analysis: A new perspective based on type inference. *Proc. Conference on Functional Programming Languages and Computer Architecture*, pp. 260–272.

Milner, R. (1978) A theory of type polymorphism in programming. *J. Comput. & Syst. Sci.*, **17**, 348–375.

Milner, R. and Tofte, M. (1991) Co-induction in relational semantics. *Theor. Comput. Sci.*, **87**(1), 209–220.

Mitchell, J. (1984) Coercion and type inference. *11th Symposium on Principles of Programming Languages*, pp. 175–185.

Mitchell, J. C. (1991) Type inference with simple subtypes. *J. Functional Programming*, **1**, 245–285.

Mossin, C. (1997) Exact flow analysis. *Proc. SAS'97, International Static Analysis Symposium: Lecture Notes in Computer Science*. Springer-Verlag.

Nielson, F. (1989) The typed lambda-calculus with first-class processes. *Proc. PARLE*, pp. 357–373.

Nielson, F. and Nielson, H. R. (1997) Infinitary control flow analysis: A collecting semantics for closure analysis. *Proc. POPL'97, 24th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pp. 332–345.

Palsberg, J. (1995) Closure analysis in constraint form. *ACM Trans. Programming Languages and Syst.*, **17**(1), 47–62. (Preliminary version in *Proc. CAAP'94, Colloquium on Trees in Algebra and Programming: Lecture Notes in Computer Science 787*, pp. 276–290. Springer-Verlag, April 1994.)

Palsberg, Je. (1998) Equality-based flow analysis versus recursive types. *ACM Trans. Programming Languages and Syst.*, **20**(6), 1251–1264.

Palsberg, J. and O'Keefe, P. M. (1995) A type system equivalent to flow analysis. *ACM Trans. Programming Languages and Systems*, **17**(4), 576–599. (Preliminary version in *Proc. POPL'95, 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pp. 367–378. San Francisco, CA, January 1995.)

Palsberg, J. and Schwartzbach, M. I. (1995) Safety analysis versus type inference. *Information & Computation*, **118**(1), 128–141.

Palsberg, J. and Smith, S. (1996) Constrained types and their expressiveness. *ACM Trans. Programming Languages and Systems*, **18**(5), 519–527.

Palsberg, J. and Zhao, T. (2000) Efficient and flexible matching of recursive types. *Proc. LICS'00, 15th Annual IEEE Symposium on Logic in Computer Science*, pp. 388–398.

Pande, H. D. and Ryder, B. G. (1996) Data-flow-based virtual function resolution. *Proc. SAS'96, International Static Analysis Symposium: Lecture Notes in Computer Science 1145*, pp. 238–254. Springer-Verlag.

Pierce, B. (1991) *Programming with intersection types, union types, and polymorphism.* Technical report CMU–CS–91–106, Carnegie Mellon University.

Plevyak, J., Zhang, X. and Chien, A. A. (1995) Obtaining sequential efficiency for concurrent object-oriented languages. *Proc. POPL'95, 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pp. 238–254.

Plotkin, G. D. (1981) *A structural approach to operational semantics.* Technical report DAIMI FN–19, Computer Science Department, Aarhus University.

Schmidt, D. (1995) Natural-semantics-based abstract interpretation. *Proc. SAS'95, International Static Analysis Symposium: Lecture Notes in Computer Science 983*, pp. 238–254. Springer-Verlag.

Sharir, M. and Pnueli, A. (1981) Two approaches to interprocedural data flow analysis. In: Muchnick, S. and Jones, N. (editors), *Program Flow Analysis, Theory and Applications.* Prentice Hall, Englewood Cliffs, New Jersey.

Shivers, O. (1991) *Control-flow analysis of higher-order languages.* PhD thesis, CMU.

Stefanescu, D. and Zhou, Y. (1994) An equational framework for flow analysis of higher-order functional programs. *Proc. ACM Conference on Lisp and Functional Programming*, pp. 318–327.

Tang, Y. M. and Jouvelot, P. (1994) Separate abstract interpretation for control-flow analysis. *Proc. TACS'94, Theoretical Aspects of Computing Software: Lecture Notes in Computer Science 789*, pp. 224–243. Springer-Verlag.

Tarski, A. (1955) A lattice-theoretical fixed point theorem and its applications. *Pacific J. Mathematics*, 285–309.

Turbak, F., Dimock, A., Muller, R. and Wells, J. B. (1997) Compiling with polymorphic and polyvariant flow types. *ACM Sigplan Workshop on Types in Compilation.* http://www.cs.bc.edu/~muller/postscript/tic97.ps.Z.

van Bakel, S. (1991) *Intersection type disciplines in lambda calculus and applicative term rewriting systems.* PhD thesis, Catholic University of Nijmegen.

Wand, M. and Steckler, P. (1994) Selective and lightweight closure conversion. *Proc. POPL'94, 21st Annual Symposium on Principles of Programming Languages*, pp. 434–445.

Wells, J. B., Dimock, A., Muller, R. and Turbak, F. (1997) A typed intermediate language for flow-directed compilation. *Proc. TAPSOFT'97, Theory and Practice of Software Development: Lecture Notes in Computer Science 1214.* Springer-Verlag.

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

Wright, D. A. (1991) A new technique for strictness analysis. *Proc. TAPSOFT'91: Lecture Notes in Computer Science 494*, pp. 235–258. Springer-Verlag.

Yi, K. (1994) Compile-time detection of uncaught exceptions in standard ML programs. *Proc. SAS'94, International Static Analysis Symposium: Lecture Notes in Computer Science 983*. Springer-Verlag.