Imperative Polymorphism by Store-Based Types as Abstract Interpretations

Casper Bach Poulsen Peter D. Mosses Paolo Torrini
Swansea University
{cscbp,p.d.mosses,p.torrini}@swansea.ac.uk

Abstract

Dealing with polymorphism in the presence of imperative features is a long-standing open problem for Hindley-Milner type systems. A widely adopted approach is the value restriction, which inhibits polymorphic generalisation and unfairly rejects various programs that cannot go wrong. We consider abstract interpretation as a tool for constructing safe and precise type systems, and investigate how to derive store-based types by abstract interpretation. We propose store-based types as a type discipline that holds potential for interesting and flexible alternatives to the value restriction.

Categories and Subject Descriptors D.3.1 [*PROGRAMMING LANGUAGES*]: Formal Definitions and Theory—semantics

Keywords type systems; operational semantics; references; polymorphism; abstract interpretation; store-based typing

1. Introduction

The Hindley-Milner type discipline is elegant and flexible for functional languages, but extending it to deal soundly with imperative features can be challenging. In particular, adding ML-style reference types to this discipline, without introducing appropriate constraints on generalisation, is well-known to break type safety [21]. Consider the program:

```
let c = ref (λx.x)
in c := (λx.1 + x);
   (!c) true
```

Assuming generalisation is unconstrained, line 1 sets `c` equal to a reference of type $\forall\alpha.\ (\alpha \to \alpha)$ `ref`. Line 2 makes an assignment which appears to be valid, since the type `int` $\to$ `int` of the expression is an instance of $\forall\alpha.\ (\alpha \to \alpha)$. Line 3 leads to a run-time type error, attempting to evaluate $1 +$ `true`. However, the static analyser does not detect the update in the store, and gives this program type `bool`, since `bool` $\to$ `bool` is an instance of $\forall\alpha.\ \alpha \to \alpha$, too.

Several techniques have been proposed to deal with this problem (see section 4.2), which may result in a serious obstacle to refactoring, hindering type-directed replacement of functional code by imperative code. The solution that has been widely adopted in practice, mainly due to its simplicity, is the so-called *value restriction* proposed by Wright [23]: generalisation of type parameters is only allowed for let-expressions where the expression being bound is a value. This turns out to be liberal enough in most cases, since unevaluated expressions can be lifted to values by η-expansion. However, this can be problematic when we are interested in program behaviour rather than merely the input-output relation, since η-lifting may force radical changes in the order of evaluation.

In this paper, we propose a novel approach to relaxing the value restriction, that we call *store-based typing*, and that we have developed using a transformational technique. The contributions we are making fall into two categories:

- Technical: we use a novel combination of techniques, including a variant of coinductive big-step semantics [11] and abstract interpretation [6] to guide the transformation of a dynamic operational semantics into a type semantics that is safe by construction.
- Practical: a novel approach to type inference with promising potential for more flexible typing disciplines for imperative polymorphism.

The rest of this paper is structured as follows: we first introduce a coinductive big-step semantics for the call-by-value λ-calculus without references in Sect. 2. The big-step semantics forms the basis for deriving safe base, mono- and polytype systems in Sect. 3, which is largely based on Cousot's work on types as abstract interpretations [5], but using an operational, rather than denotational, approach. Extending our language with references in Sect. 4, we propose a store-based polytype semantics for imperative let-polymorphism. Section 5 recalls related work and outlines future directions.

2. A Novel Style of Coinductive Semantics

Consider the following grammar for the λ-calculus with integer constants $i \in \mathbb{Z}$ and variables $x \in \mathit{Var}$.

$$\mathit{Expr} \ni e ::= \lambda x.e \mid e\ e \mid x \mid i$$

Using ordinary big-step semantics, also known as *natural semantics* [10], the *inductive* interpretation of the following rules defines the judgment $\rho \vdash e \Rightarrow v$ to hold just when left to right, eager evaluation of e in environment ρ terminates with value v.

$$\frac{}{\rho \vdash \lambda x.e \Rightarrow \langle x, e, \rho\rangle} \qquad \frac{\rho(x) = v}{\rho \vdash x \Rightarrow v} \qquad \frac{}{\rho \vdash i \Rightarrow i}$$

$$\frac{\rho \vdash e_1 \Rightarrow \langle x, e, \rho'\rangle \quad \rho \vdash e_2 \Rightarrow v_2 \quad \rho'[x \mapsto v_2] \vdash e \Rightarrow v}{\rho \vdash e_1\ e_2 \Rightarrow v}$$

Here, environments $\rho \in \mathit{Env} \triangleq \mathit{Var} \xrightarrow{\text{fin}} \mathit{Val}$, and values $v \in \mathit{Val}$ include integers i and closures $\langle x, e, \rho \rangle$.

The above semantics does not distinguish between expressions whose evaluation diverges (e.g., $\omega \triangleq (\lambda x.x\,x)(\lambda x.x\,x)$) and those whose evaluation gets stuck (e.g., $0\,0$). Leroy and Grall [11] show that the *coinductive* interpretation of such big-step rules lets some diverging expressions have values, but not all. For example, $\rho \vdash \omega \Rightarrow v$ for all ρ, v, but not $\rho \vdash \omega\,(0\,0) \Rightarrow v$, since the premises of the application rule require a value for the stuck term

To detect divergence one can follow Cousot [7] as well as Leroy and Grall, and define a divergence predicate $\Rightarrow^\infty$ coinductively:

$$\frac{\rho \vdash e_1 \Rightarrow^\infty}{\rho \vdash e_1\ e_2 \Rightarrow^\infty} \qquad \frac{\rho \vdash e_1 \Rightarrow \langle x, e, \rho' \rangle \quad \rho \vdash e_2 \Rightarrow^\infty}{\rho \vdash e_1\ e_2 \Rightarrow^\infty}$$

$$\frac{\rho \vdash e_1 \Rightarrow \langle x, e, \rho' \rangle \quad \rho \vdash e_2 \Rightarrow v_2 \quad \rho'[x \mapsto v_2] \vdash e \Rightarrow^\infty}{\rho \vdash e_1\ e_2 \Rightarrow^\infty}$$

This gives $\rho \vdash \omega\,(0\,0) \Rightarrow^\infty$. There are two pragmatic problems: (i) the need for extra rules and premises, known as the *duplication problem* [3, 4]; (ii) to prove a property of our semantics may involve *both* $\Rightarrow$ and $\Rightarrow^\infty$. The alternative of merely adding a $\perp$ element to the semantics still involves a significant number of extra rules and premises that also clutter the reasoning steps involved in proofs.

Here, we propose a new way of encoding divergence that subsumes both the ordinary $\rho \vdash e \Rightarrow v$ relation and the $\rho \vdash e \Rightarrow^\infty$ predicate. We introduce a 'divergence flag':

$$\mathit{Div} \ni \delta ::= \downarrow \mid \uparrow$$

A change from $\downarrow$ to $\uparrow$ corresponds to divergence arising. Augmenting our relation with this flag, the judgment becomes $\rho \vdash e_{/\delta} \Rightarrow v_{/\delta'}$, saying that evaluating e in divergence state δ gives the outcome v in divergence state δ'. Figure 1 gives the rules for our augmented relation. The crucial rule that allows us to propagate divergence between premises is the divergence rule DIV. The intuition is that, if we are diverging, no value is produced, so we may choose any value v. The other rules are obtained from the original big-step rules systematically, reflecting the intended order of evaluation.

$$\frac{}{\rho \vdash i_{/\downarrow} \Rightarrow i_{/\downarrow}}\ (\textsc{Int}) \qquad \frac{}{\rho \vdash e_{/\uparrow} \Rightarrow v_{/\uparrow}}\ (\textsc{Div})$$

$$\frac{\rho(x) = v}{\rho \vdash x_{/\downarrow} \Rightarrow v_{/\downarrow}}\ (\textsc{Var}) \qquad \frac{}{\rho \vdash \lambda x.e_{/\downarrow} \Rightarrow \langle x, e, \rho \rangle_{/\downarrow}}\ (\textsc{Abs})$$

$$\frac{\rho \vdash e_{1/\downarrow} \Rightarrow \langle x, e, \rho' \rangle_{/\delta} \qquad \rho \vdash e_{2/\delta} \Rightarrow v_{2/\delta'} \qquad \rho'[x \mapsto v_2] \vdash e_{/\delta'} \Rightarrow v_{/\delta''}}{\rho \vdash e_1\ e_{2/\downarrow} \Rightarrow v_{/\delta''}}\ (\textsc{App})$$

Figure 1. Coinductive evaluation rules using a divergence flag

It is worth spelling out the relationship between the rules with the divergence flag and Leroy and Grall's big-step semantics. The rules corresponding to the $\Rightarrow^\infty$ relation can be obtained by unfolding the premises of the big-step rules relative to the DIV rule. The judgment $\rho \vdash e_{/\downarrow} \Rightarrow v_{/\uparrow}$ is not derivable under an inductive interpretation of our rules (ensured by insisting that the conclusion and first premise start in a converging state for all rules, except DIV). Therefore, if we can derive it using coinduction, it must be a diverging computation. It is, however, possible to prove $\rho \vdash e_{/\downarrow} \Rightarrow v_{/\downarrow}$ for some diverging computations (e.g., $\rho \vdash \omega_{/\downarrow} \Rightarrow v_{/\downarrow}$ for any v).

In summary, a proof that something coevaluates to $\downarrow$ does not imply that it is in the inductive relation. In contrast, a proof that something coevaluates to $\uparrow$ implies that it is a diverging computation, and thus not in the inductive relation. The lack of distinction between some converging and diverging computations in the coinductive relation is not important for the purpose of this paper. The important point is that we have obtained a concisely defined relation containing all converging and diverging computations.

3. Deriving Church/Curry Polytypes

The abstract interpretation framework [6] provides a systematic method for constructing safe approximations of the semantics of programs. We follow [5] in abstracting type systems from the so-called collecting semantics, but start from coinductive big-step operational semantics instead of denotational semantics.

We first define an abstraction to *base types* (obtained by replacing types for ground values in expressions) as in [13, 14]. Subsequently, we introduce an abstraction of base types (after extending the language with a let construct) to Hindley-Milner polytypes. Polytypes can be further abstracted to monotypes, following [5].

3.1 Collecting Semantics

Let $\mathbb{S} \triangleq \mathit{Env} \to \wp(\mathit{Val} \times \mathit{Div})$. With reference to the coinductive interpretation of the rules in Fig. 1, let $\mathbf{S}[\![\bullet]\!] \in \mathit{Expr} \to \mathbb{S}$ be defined by $\mathbf{S}[\![e]\!] \triangleq \Lambda\rho.\ \{\langle v, \delta \rangle \mid \rho \vdash e_{/\downarrow} \Rightarrow v_{/\delta}\}$, where Λ is meta-level function abstraction (following [5]).

A *property* of an expression is a subset of $\mathbb{S}$, i.e., an element of $\mathbb{C} \triangleq \wp(\mathbb{S})$. We are interested in the programs that do not get stuck. However, this property is undecidable. It can be approximated by introducing a notion of type. The *collecting semantics* $\mathbf{C}[\![\bullet]\!] \in \mathit{Expr} \to \mathbb{C}$, defined by $\mathbf{C}[\![e]\!] \triangleq \{\mathbf{S}[\![e]\!]\}$, is the property that gives the most precise information on a program. Abstract interpretation gives us a method for constructing approximations of well-behaved programs as type systems, starting from the collecting semantics, by using abstractions that ensure type soundness by definition. The notion of type soundness we are interested in here is the usual one, adapted to our big-step semantics:

$$\Gamma \vdash e : t \Longrightarrow \vdash \rho : \Gamma \Longrightarrow \exists v, \delta.\ \vdash v : t \wedge \rho \vdash e_{/\downarrow} \Rightarrow v_{/\delta}$$

where Γ is a typing environment and t a type.

Given two partially ordered sets $\langle \mathbb{P}, \sqsubseteq^{\mathbb{P}} \rangle$, $\langle \mathbb{Q}, \sqsubseteq^{\mathbb{Q}} \rangle$, a Galois connection $\langle \mathbb{P}\ \sqsubseteq^{\mathbb{P}} \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{Q}, \sqsubseteq^{\mathbb{Q}} \rangle$ arises for total functions $\alpha \in \mathbb{P} \to \mathbb{Q}$ (*abstraction*) and $\gamma \in \mathbb{Q} \to \mathbb{P}$ (*concretisation*) whenever $\alpha(p) \sqsubseteq^{\mathbb{Q}} q \iff p \sqsubseteq^{\mathbb{P}} \gamma(q)$ for all $p \in \mathbb{P}$ and $q \in \mathbb{Q}$. The progression we establish can be depicted:[1]

$$\langle \mathbb{C}, \subseteq^{\mathbb{C}} \rangle \underset{\alpha^b}{\overset{\gamma^b}{\leftrightarrows}} \langle \mathbb{B}, \supseteq^{\mathbb{B}} \rangle \underset{\alpha^p}{\overset{\gamma^p}{\leftrightarrows}} \langle \mathbb{P}, \supseteq^{\mathbb{P}} \rangle \underset{\alpha^m}{\overset{\gamma^m}{\leftrightarrows\!\!\!\!\rightarrow}} \langle \mathbb{M}, \supseteq^{\mathbb{M}} \rangle$$

where $\mathbb{C}$ is the domain for the collecting semantics, $\mathbb{B}$ for the base type semantics, $\mathbb{P}$ for the polytype semantics, and $\mathbb{M}$ for the monotype semantics. At each step, abstraction is defined so as to induce a typing relation as an abstract derivation that is sound with respect to the concrete one. The Galois connections arising by composition, from $\alpha^p \circ \alpha^b$ and $\alpha^m \circ \alpha^p \circ \alpha^b$ respectively, ensure that type soundness holds for polytypes and monotypes.

3.2 Base Type Abstraction

The idea is to replace the set of integers by an `int` type and preserve the structure of all other constructs. We proceed in three steps: defining the domain, describing the Galois connection, and using this to infer the structure of the type semantics.

Domain definitions. We are interested in the type of values that a program returns when it terminates. For this reason, we do not need the Div flag in the definition of the base type domain:

$$\mathit{BType} \ni b ::= \texttt{int} \mid \langle x, e, \rho^b \rangle$$

$$\rho^b \in \mathit{BEnv} \triangleq \mathit{Var} \xrightarrow{\text{fin}} \mathit{BType} \qquad \mathbb{B} \triangleq \wp(\mathit{BEnv} \times \mathit{BType})$$

[1] The double headed arrow represents *Galois insertions*, meaning that the mapping is surjective.

Galois connection. We define the abstraction α^b of the collecting semantics in terms of an abstraction α^b_s of the denotations, which in turn is defined in terms of α^b_v which abstracts values, and α^b_ρ which abstracts environments by pointwise application of α^b_v.

$$\begin{array}{rr} \alpha^b \in \mathbb{C} \to \mathbb{B} & \alpha^b_s \in \mathbb{S} \to \mathbb{B} \\ \alpha^b_v \in \mathit{Val} \to \mathit{BType} & \alpha^b_\rho \in \mathit{Env} \to \mathit{BEnv} \\ \alpha^b_v(i) \triangleq \texttt{int} & \alpha^b_v(\langle x, e, \rho\rangle) \triangleq \langle x, e, \alpha^b_\rho(\rho)\rangle \end{array}$$

$$\alpha^b_s(S) \triangleq \{\langle \rho^b, b\rangle \mid \forall \rho.\ \rho^b = \alpha^b_\rho(\rho) \Longrightarrow \exists v.\ b = \alpha^b_v(v) \wedge \langle v, \downarrow\rangle \in S(\rho)\}$$

$$\alpha^b_\rho(\rho) \triangleq \Lambda x.\ \alpha^b_v(\rho(x)) \qquad \alpha^b(C) \triangleq \bigcap_{S\in C}^{\mathbb{B}} \alpha^b_s(S)$$

The definition of α^b_s is designed to match our soundness schema. However, here it is convenient to strengthen it by restricting typeability to terminating programs, in order to derive the typing relation as an inductive one. Notice that from this particular definition of α^b_s, it follows that diverging and stuck programs have no type.

For all $\Delta \subseteq \mathbb{C}$, the following property holds:

$$\alpha^b(\bigcup^{\mathbb{C}} \Delta) = \bigcap_{C\in\Delta}^{\mathbb{B}} \alpha^b(C)$$

since:

$$\alpha^b(\bigcup^{\mathbb{C}} \Delta) = \bigcap_{S\in\bigcup\Delta}^{\mathbb{B}} \alpha^b_s(S) = \bigcap_{C\in\Delta, S\in C}^{\mathbb{B}} \alpha^b_s(S) = \bigcap_{C\in\Delta}^{\mathbb{B}} \alpha^b(C)$$

which suffices to establish the Galois connection:

$$\langle \mathbb{C}, \subseteq^{\mathbb{C}}\rangle \underset{\alpha^b}{\overset{\gamma^b}{\leftrightarrows}} \langle \mathbb{B}, \supseteq^{\mathbb{B}}\rangle$$

Type semantics. Given the abstract specification of the desired typing semantics $\mathbf{B}[\![\bullet]\!] \in \mathit{Expr} \to \mathbb{B}$:

$$\mathbf{B}[\![e]\!] \triangleq \{\langle \rho^b, b\rangle \mid \rho^b \vdash e \Rightarrow^b b\}$$

the Galois connection provides a guideline for inferring its structural definition. The guiding constraint is the following:

$$\alpha^b(\mathbf{C}[\![e]\!]) \supseteq^{\mathbb{B}} \mathbf{B}[\![e]\!]$$

and therefore:

$$\alpha^b_s(\mathbf{S}[\![e]\!]) \supseteq^{\mathbb{B}} \mathbf{B}[\![e]\!]$$

Unfolding the definitions of $\mathbf{B}[\![\bullet]\!]$, $\mathbf{S}[\![\bullet]\!]$, and α^b_s, we obtain:

$$\rho^b \vdash e \Rightarrow^b b \Longrightarrow \rho^b = \alpha^b_\rho(\rho) \Longrightarrow \exists v.\ b = \alpha^b_v(v) \wedge \rho \vdash e_{/\downarrow} \Rightarrow v_{/\downarrow}$$

Reasoning on the structure of the $\Rightarrow$ relation, a definition of $\Rightarrow^b$ can be found, by choosing it so that the proof goes through by induction on $\Rightarrow^b$. The simplest relation that admits such a structural induction proof is the one given by the following rules:

$$\frac{}{\rho^b \vdash i \Rightarrow^b \texttt{int}}\ (\textsc{B-Int}) \qquad \frac{\rho^b(x) = b}{\rho^b \vdash x \Rightarrow^b b}\ (\textsc{B-Var})$$

$$\frac{}{\rho^b \vdash \lambda x.e \Rightarrow^b \langle x, e, \rho^b\rangle}\ (\textsc{B-Abs})$$

$$\frac{\rho^b \vdash e_1 \Rightarrow^b \langle x, e, \rho^b\rangle \qquad \rho^b \vdash e_2 \Rightarrow^b b_2 \qquad \rho^b[x \mapsto b_2] \vdash e \Rightarrow^b b_1}{\rho^b \vdash e_1\, e_2 \Rightarrow^b b_1}\ (\textsc{B-App})$$

3.3 Hindley-Milner Polymorphism

Determining the base type of an expression essentially involves evaluating the expression, hence base types are rather useless as a type system. However, they can be useful as an intermediate abstraction [14]. Notice that base types have an implicitly polymorphic character (closure types can be informally understood to be polymorphic). For this reason, they provide an optimal starting point to define abstractions to proper polymorphic types.

Here we are interested in Hindley-Milner polymorphism, which has a well-known syntactic characterisation with good computational properties [8]. Also known as let-polymorphism, it is syntactically an extension of typed λ-calculus with type variables, allowing for type schemes (i.e., type expressions with variables), and it does not require any explicit use of quantification. Semantically, we characterise types as monotypes, and explain type schemes away in terms of sets of monotypes, that for us are the polytypes.

Hindley-Milner polymorphism is more restrictive than the unconstrained syntactic polymorphism of system F, but is more liberal than so-called weak polymorphism [17]. Weak polymorphism does not allow instantiation of type variables with polytypes (*predicative* restriction), nor functions to have polytype arguments (*prenex* restriction). Hindley-Milner polymorphism is obtained by relaxing the prenex restriction in let-expressions, whose evaluation is characterised by the following rule:

$$\frac{\rho \vdash e_{2/\downarrow} \Rightarrow v_{2/\delta} \qquad \rho[x \mapsto v_2] \vdash e_{1/\delta} \Rightarrow v_{1/\delta'}}{\rho \vdash (\texttt{let}\ x\ \texttt{=}\ e_2\ \texttt{in}\ e_1)_{/\downarrow} \Rightarrow v_{1/\delta'}}\ (\textsc{Let})$$

Domain definitions. We can abstractly specify the polytype domain as follows, on top of a minimal monotype syntax.

$$\mathit{MType} \ni m ::= \texttt{int} \mid m \to m$$

$$p \in \mathit{PType} \triangleq \wp(\mathit{MType})$$

$$\rho^p \in \mathit{PEnv} \triangleq \mathit{Var} \xrightarrow{\text{fin}} \mathit{PType} \qquad \mathbb{P} \triangleq \wp(\mathit{PEnv} \times \mathit{MType})$$

Galois connection. Value and environment abstraction give us naturally sets (comparable to the general types in [14]). Such sets may not be computable, but this does not prevent us from specifying type soundness by abstraction.

$$\begin{array}{rr} \alpha^p \in \mathbb{B} \to \mathbb{P} & \alpha^p_\rho \in \mathit{BEnv} \to \wp(\mathit{PEnv}) \\ \alpha^p_b \in \mathit{BType} \to \mathit{PType} & \\ \alpha^p_b(\texttt{int}) \triangleq \{\texttt{int}\} & \end{array}$$

$$\alpha^p_b(\langle x, e, \rho^b\rangle) \triangleq \{m_2 \to m_1 \mid \forall b_2.\ m_2 \in \alpha^p_b(b_2) \Longrightarrow \exists b_1.\ m_1 \in \alpha^p_b(b_1) \wedge \rho^b[x \mapsto b_2] \vdash e \Rightarrow^b b_1\}$$

$$\alpha^p_\rho(\rho^b) \triangleq \{\rho^p \mid \forall x.\ \rho^p(x) \subseteq \alpha^p_b(\rho^b(x))\}$$

$$\alpha^p(B) \triangleq \{\langle \rho^p, m\rangle \mid \forall \rho^b.\ \rho^p \in \alpha^p_\rho(\rho^b) \Longrightarrow \exists b.\ m \in \alpha^p_b(b) \wedge \langle \rho^b, b\rangle \in B\}$$

The following Galois connection holds between base types and polytypes:

$$\langle \mathbb{B}, \supseteq^{\mathbb{B}}\rangle \underset{\alpha^p}{\overset{\gamma^p}{\leftrightarrows}} \langle \mathbb{P}, \supseteq^{\mathbb{P}}\rangle$$

Type semantics. The specification of the semantics is:

$$\mathbf{P}[\![\bullet]\!] \in \mathit{Expr} \to \mathbb{P}$$

$$\mathbf{P}[\![e]\!] \triangleq \{\langle \rho^p, m\rangle \mid \rho^p \vdash e \Rightarrow^p m\}$$

Since Hindley-Milner polymorphism has principal types [8], we expect principality to be provable along the lines of [5, Sect. 12].

The guiding constraint provided by the Galois connection to define $\Rightarrow^p$ is the following:

$$\alpha^p(\mathbf{B}[\![e]\!]) \supseteq^{\mathbb{B}} \mathbf{P}[\![e]\!]$$

and therefore:

$$\forall m \in p.\ \rho^p \vdash e \Rightarrow^p m \Longrightarrow \rho^p \in \alpha^p_\rho(\rho^b) \Longrightarrow$$
$$\exists b.\ p \subseteq \alpha^p_b(b) \wedge \rho^b \vdash e \Rightarrow^b b$$

The expected definition of $\Rightarrow^p$ can be found by reasoning on the structure of e in order to make the proof go through by induction on $\Rightarrow^p$. Rules P-INT and P-VAR are similar to B-INT and B-VAR. The rule for $\rightarrow$ introduction:

$$\frac{\rho^p[x \mapsto \{m_2\}] \vdash e \Rightarrow^p m_1}{\rho^p \vdash \lambda x.e \Rightarrow^p m_2 \rightarrow m_1} \quad \text{(P-ABS)}$$

is basically built into the definition of α^p_b. Notice that x has to be typed by m_2 in this rule as b_2 can vary arbitrarily in that definition. The restriction of the polytype of x to a singleton here corresponds to the fact that the prenex restriction applies to non-let-expressions. The rule for $\rightarrow$ elimination:

$$\frac{\rho^p \vdash e_1 \Rightarrow^p m_2 \rightarrow m_1 \qquad \rho^p \vdash e_2 \Rightarrow^p m_2}{\rho^p \vdash e_1\ e_2 \Rightarrow^p m_1} \quad \text{(P-APP)}$$

can be inferred from the case of $e = e_1\, e_2$ of the proof. In fact, it is the simplest rule such that the premises of the B-APP instance needed to get the statement conclusion can be obtained via the induction hypothesis: given $\rho^p \in \alpha^p_\rho(\rho^b)$, from $\rho^p \vdash e_2 \Rightarrow^p m_2$ it follows $m_2 \in \alpha^p_b(b') \wedge \rho^b \vdash e_2 \Rightarrow^b b'$ for some b'; from $\rho^p \vdash e_1 \Rightarrow^p m_2 \rightarrow m_1$ it follows $(m_2 \rightarrow m_1) \in \alpha^p_b(\langle x, e', \rho^b\rangle) \wedge \rho^b \vdash e_1 \Rightarrow^b \langle x, e', \rho^b\rangle$ for some e', and thus, by definition of α^p_b, also $m_1 \in \alpha^p_b(b) \wedge \rho^b[x \mapsto b'] \vdash e' \Rightarrow^b b$ for some b. The rule for let:

$$\frac{p \neq \emptyset \qquad \begin{array}{l}\forall m_2 \in p.\ \rho^p \vdash e_2 \Rightarrow^p m_2 \\ \rho^p[x \mapsto p] \vdash e_1 \Rightarrow^p m_1\end{array}}{\rho^p \vdash \texttt{let}\ x\ \texttt{=}\ e_2\ \texttt{in}\ e_1 \Rightarrow^p m_1} \quad \text{(P-LET)}$$

can be inferred from the case $e = \texttt{let}\ x\ \texttt{=}\ e_2\ \texttt{in}\ e_1$. First we need to extend the base type abstraction to LET. This gives us:

$$\frac{\rho^b \vdash e_2 \Rightarrow^b b_2 \qquad \rho^b[x \mapsto b_2] \vdash e_1 \Rightarrow^b b_1}{\rho^b \vdash \texttt{let}\ x\ \texttt{=}\ e_2\ \texttt{in}\ e_1 \Rightarrow^b b_1} \quad \text{(B-LET)}$$

Rule P-LET can then be justified as the rule needed for the B-LET case: from the first premise of P-LET it follows there is a b_2 such that $p \subseteq \alpha^p_b(b_2) \wedge \rho^b \vdash e_2 \Rightarrow^b b_2$; from $\rho^p[x \mapsto p] \vdash e_1 \Rightarrow^p m_1$ it follows $m_1 \in \alpha^p_b(b_1) \wedge \rho^b[x \mapsto b_2] \vdash e' \Rightarrow^b b_1$.

3.4 Church/Curry Monotype Abstraction

Following Cousot [5], we can specify the monotype domain:

$$\rho^m \in \mathit{MEnv} \triangleq \mathit{Var} \xrightarrow{\text{fin}} \mathit{MType} \qquad \mathbb{M} \triangleq \wp(\mathit{MEnv} \times \mathit{MType})$$

and define a Galois insertion between polytypes and monotypes:

$$\langle \mathbb{P}, \supseteq^{\mathbb{P}}\rangle \underset{\alpha^m}{\overset{\gamma^m}{\leftrightarrows}} \langle \mathbb{M}, \supseteq^{\mathbb{M}}\rangle$$

$$\alpha^m(P) \triangleq \{(\rho^m, m) \mid (\Lambda x.\{\rho^m(x)\}, m) \in P\}$$
$$\gamma^m(M) \triangleq \{(\Lambda x.\{\rho^m(x)\}, m) \mid (\rho^m, m) \in M\}$$

The surjectivity of α^m ensures γ^m is injective, but α^m involves a loss of information [5, Sect. 7]. The monotype semantics is:

$$\mathbf{M}[\![\bullet]\!] \in \mathit{Expr} \rightarrow \mathbb{M}$$
$$\mathbf{M}[\![e]\!] \triangleq \{\langle \rho^m, m\rangle \mid \rho^m \vdash e \Rightarrow^m m\}$$

By following the constraint:

$$\alpha^m(\mathbf{P}[\![e]\!]) \supseteq^{\mathbb{B}} \mathbf{M}[\![e]\!]$$

it is not difficult to see that the definition of $\Rightarrow^m$ simply involves replacing polytype environments with monotype ones in the $\Rightarrow^p$ rules, leaving the rest unchanged. Notice that $\texttt{let}\ x\ \texttt{=}\ e_2\ \texttt{in}\ e_1$ and $(\lambda x.e_1)\, e_2$ here become equivalent.

4. Store-Based Types

In this section we propose a novel approach to imperative polymorphism. It is straightforward to extend the language considered in the previous section with ML-style references, and to add locations and stores to its coinductive big-step evaluation rules. When using abstract interpretation to define type systems for the extended language, however, we need to choose how to approximate the locations and stores.

Here, we give a polymorphic type system that results from retaining exact information about allocation and updates, while approximating all values other than locations by types (as in the previous section). This system allows so-called *strong updates* [2] where assignment can change the type of value stored at a location. We give examples of programs that have types in our system, but not with previous approaches. Further exploration of our approach is needed to establish its potential usefulness and safety.

4.1 Store-Based Types

Adding stores to the call-by-value λ-calculus, the grammar from Sect. 2 is extended as follows:

$$\mathit{Expr} \ni e ::= \ldots \mid \texttt{ref}\ e \mid \texttt{!}e \mid e\ \texttt{:=}\ e$$

Values $v \in \mathit{Val}$ now include locations $l \in \mathit{Loc}$. The rules in Fig. 2 give the dynamic semantics of our extended language, which defines the judgment $\rho \vdash e_{/\sigma|\delta} \Rightarrow v_{/\sigma'|\delta'}$ to hold when left to right evaluation of expression e in environment ρ, store σ, and divergence state δ, gives the outcome consisting of value v, result store σ', and divergence state δ'. Here, stores $\sigma \in \mathit{Loc} \rightarrow \mathit{Val}$ denote possibly-infinite maps. In addition to the rules R-REF, R-DREF, and R-ASGN, we have added the rule R-STO-WK. The latter rule is used to allow coinductive reasoning about diverging computations that produce infinite stores [16] (for example, it provides a means of proving that $(\lambda x.x\,x)(\lambda x.\texttt{let}\ r\ \texttt{=}\ \texttt{ref}\ 1\ \texttt{in}\ x\,x)$ diverges; without it, a coinductive hypothesis will not match).

A *store-based type* is a pair consisting of a type and a type store. The syntax of types is:

$$\mathit{MType}^\varsigma \ni M ::= \texttt{int} \mid l \mid \langle M, \varsigma\rangle \rightarrow \langle M, \varsigma\rangle$$

Here, $\varsigma \in \mathit{Loc} \xrightarrow{\text{fin}} \mathit{MType}^\varsigma$ is a type store.[2] Store-based types depart from the usual approach to store typing [17]: we introduce the notion of type store as the type level abstraction of a store, in which locations are treated as types themselves. Store-based function types $\langle M, \varsigma\rangle \rightarrow \langle M', \varsigma'\rangle$ record an argument type store ς and a return type store ς'. The rules in Fig. 3 define the judgment $\Gamma^P \vdash e_{/\varsigma} \Rightarrow^S M_{/\varsigma'}$ to hold when expression e in the polytype environment $\Gamma^P \in \mathit{Var} \xrightarrow{\text{fin}} \wp(\mathit{MType}^\varsigma)$ and type store ς has type M in type store ς'.

We highlight rules that differ from traditional ML typing rules:

S-ABS. The function type records the inferred argument type M_2, argument type store ς_0 under which the function body evaluates, the result type M_1, and the updated type store ς_Δ resulting from function body evaluation.

S-APP. Applying a store-based abstraction involves checking that the current store ς is compatible with the function argument type store ς_0, using the $\preceq^\varsigma$ relation. When both type store and argument type are compatible with the function type, the type store ς'' is updated relative to the function return type store ς_Δ, using the $\oslash$ operation, which shadows mappings in ς'' by ς_Δ, and creates fresh locations for newly allocated references between ς'' and ς_Δ.

[2] Whereas dynamic stores are potentially infinite, type stores are finite. Typing restricts stores to those that are finitely typeable.

$$\frac{}{\rho \vdash i_{/\sigma|\downarrow} \Rightarrow i_{/\sigma|\downarrow}}\ \text{(R-INT)} \qquad \frac{}{\rho \vdash e_{/\sigma|\uparrow} \Rightarrow v_{/\sigma'|\uparrow}}\ \text{(R-DIV)}$$

$$\frac{\rho(x) = v}{\rho \vdash x_{/\sigma|\downarrow} \Rightarrow v_{/\sigma|\downarrow}}\ \text{(R-VAR)}$$

$$\frac{\rho \vdash e_{/\sigma|\delta} \Rightarrow v_{/\sigma'|\delta'} \quad l \notin (\mathrm{dom}(\sigma) \cup \mathrm{dom}(\sigma'))}{\rho \vdash e_{/\sigma[l \mapsto v']|\delta} \Rightarrow v_{/\sigma'[l \mapsto v']|\delta'}}\ \text{(R-STO-WK)}$$

$$\frac{}{\rho \vdash \lambda x.e_{/\sigma|\downarrow} \Rightarrow \langle x, e, \rho \rangle_{/\sigma|\downarrow}}\ \text{(R-ABS)}$$

$$\frac{\rho \vdash e_{1/\sigma|\downarrow} \Rightarrow \langle x, e, \rho' \rangle_{/\sigma'|\delta} \quad \rho \vdash e_{2/\sigma'|\delta} \Rightarrow v_{2/\sigma''|\delta'} \quad \rho'[x \mapsto v_2] \vdash e_{/\sigma''|\delta'} \Rightarrow v_{/\sigma'''|\delta''}}{\rho \vdash e_1\ e_{2/\sigma|\downarrow} \Rightarrow v_{/\sigma'''|\delta''}}\ \text{(R-APP)}$$

$$\frac{\rho \vdash e_{2/\downarrow|\sigma} \Rightarrow v_{2/\delta|\sigma'} \quad \rho[x \mapsto v_2] \vdash e_{/\delta|\sigma'} \Rightarrow v_{/\delta'|\sigma''}}{\rho \vdash \texttt{let}\ x\ \texttt{=}\ e_2\ \texttt{in}\ e_{1/\downarrow|\sigma} \Rightarrow v_{/\delta'|\sigma''}}\ \text{(R-LET)}$$

$$\frac{\rho \vdash e_{/\sigma|\downarrow} \Rightarrow v_{/\sigma'|\delta} \quad l \notin \mathrm{dom}(\sigma')}{\rho \vdash \texttt{ref}\ e_{/\sigma|\downarrow} \Rightarrow l_{/\sigma'[l \mapsto v]|\delta}}\ \text{(R-REF)}$$

$$\frac{\rho \vdash e_{/\sigma|\downarrow} \Rightarrow l_{/\sigma'|\delta}}{\rho \vdash \texttt{!}e_{/\sigma|\downarrow} \Rightarrow \sigma'(l)_{/\sigma'|\delta}}\ \text{(R-DREF)}$$

$$\frac{\rho \vdash e_{1/\sigma|\downarrow} \Rightarrow l_{/\sigma'|\delta} \quad l \in \mathrm{dom}(\sigma') \quad \rho \vdash e_{2/\sigma'|\delta} \Rightarrow v_{2/\sigma''|\delta'}}{\rho \vdash e_1\ \texttt{:=}\ e_{2/\sigma|\downarrow} \Rightarrow v_{2/\sigma''[l \mapsto v_2]|\delta'}}\ \text{(R-ASGN)}$$

Figure 2. Coinductive big-step rules for call-by-value λ-calculus with references

S-REF **and** S-DREF**.** Locations are reflected at the type-level. Dereferencing a location via S-DREF produces the type stored at the corresponding location in a type store ς.

S-ASGN**.** Assignment supports strong updates: the type being assigned to a location is not checked against the type assigned to the location in the type store before the update.

4.2 Store-Based Types for Imperative Polymorphic Type Inference

We consider examples of how store-based types allow for Hindley-Milner polymorphism. Using the rules in Fig. 3, $\lambda x.\texttt{ref}\ x$ can be assigned a type in the set $\{\langle M, \cdot \rangle \to \langle l, (l \mapsto M) \rangle \mid M \in \mathit{MType}^{\varsigma} \wedge l \in \mathit{Loc}\}$, where $\cdot$ is an empty map. Extending our language with sequencing and booleans it follows that:

$$\mathit{mkref} \triangleq \texttt{let}\ m\ \texttt{=}\ (\lambda x.\texttt{ref}\ x)\ \texttt{in}\ m\,1\texttt{;}\ m\,\texttt{true}$$
$$\cdot \vdash \mathit{mkref}_{/\cdot} \Rightarrow^S l_{/(l \mapsto \texttt{bool})}$$

The following expression which adds an application of the identity function inside the bound let-expression produces the same type and type store as mkref:

$$\mathit{mkref}' \triangleq \texttt{let}\ m\ \texttt{=}\ (\lambda y.y)\,(\lambda x.\texttt{ref}\ x)\ \texttt{in}\ m\,1\texttt{;}\ m\,\texttt{true}$$

Consider the following somewhat contrived expression from [22]:

$$\begin{aligned}\mathit{effect} \triangleq\ & \lambda z.\texttt{let}\ \mathit{id}\ \texttt{=}\ (\lambda x.\texttt{if true then}\ z \\ & \qquad \texttt{else}\ (\lambda y.\texttt{ref}\ x\texttt{;}\ y)\texttt{;}\ x) \\ & \texttt{in}\ \mathit{id}\,1\texttt{;}\ \mathit{id}\,\texttt{true}\end{aligned}$$

If we further extend our language with conditionals, the judgment $\cdot \vdash \mathit{effect}_{/\cdot} \Rightarrow^S \texttt{bool}_{/\cdot}$ holds. Since our type system supports

$$\frac{}{\Gamma^P \vdash i_{/\varsigma} \Rightarrow^S \texttt{int}_{/\varsigma}}\ \text{(S-INT)}$$

$$\frac{\Gamma^P(x) = P \quad M \in P}{\Gamma^P \vdash x_{/\varsigma} \Rightarrow^S M_{/\varsigma}}\ \text{(S-VAR)}$$

$$\frac{\Gamma^P[x \mapsto \{M_2\}] \vdash e_{/\varsigma_0} \Rightarrow^S M_{1/\varsigma_\Delta}}{\Gamma^P \vdash \lambda x.e_{/\varsigma} \Rightarrow^S \langle M_2, \varsigma_0 \rangle \to \langle M_1, \varsigma_\Delta \rangle_{/\varsigma}}\ \text{(S-ABS)}$$

$$\frac{\Gamma^P \vdash e_{1/\varsigma} \Rightarrow^S \langle M_2, \varsigma_0 \rangle \to \langle M_1, \varsigma_\Delta \rangle_{/\varsigma'} \quad \Gamma^P \vdash e_{2/\varsigma'} \Rightarrow^S M_{2/\varsigma''} \quad \varsigma_0 \preceq \varsigma''}{\Gamma^P \vdash e_1\ e_{2/\varsigma} \Rightarrow^S M_{1/\varsigma'' \oslash \varsigma_\Delta}}\ \text{(S-APP)}$$

$$\frac{P \neq \emptyset \qquad \forall M_2 \in P.\, \Gamma^P \vdash e_{2/\varsigma} \Rightarrow^S M_{2/\varsigma'} \quad \Gamma^P[x \mapsto P] \vdash e_{1/\varsigma'} \Rightarrow^S M_{1/\varsigma''}}{\Gamma^P \vdash \texttt{let}\ x\ \texttt{=}\ e_2\ \texttt{in}\ e_{1/\varsigma} \Rightarrow^S M_{1/\varsigma''}}\ \text{(S-LET)}$$

$$\frac{\Gamma^P \vdash e_{/\varsigma} \Rightarrow^S M_{/\varsigma'} \quad l \notin \mathrm{dom}(\varsigma')}{\Gamma^P \vdash \texttt{ref}\ e_{/\varsigma} \Rightarrow^S l_{/\varsigma'[l \mapsto M]}}\ \text{(S-REF)}$$

$$\frac{\Gamma^P \vdash e_{/\varsigma} \Rightarrow^S l_{/\varsigma'}}{\Gamma^P \vdash \texttt{!}e_{/\varsigma} \Rightarrow^S \varsigma'(l)_{/\varsigma'}}\ \text{(S-DREF)}$$

$$\frac{\Gamma^P \vdash e_{1/\varsigma} \Rightarrow^S l_{/\varsigma'} \quad l \in \mathrm{dom}(\varsigma') \quad \Gamma^P \vdash e_{2/\varsigma'} \Rightarrow^S M_{2/\varsigma''}}{\Gamma^P \vdash e_1\ \texttt{:=}\ e_{2/\varsigma} \Rightarrow^S M_{2/\varsigma''[l \mapsto M_2]}}\ \text{(S-ASGN)}$$

$$\begin{aligned}\varsigma_1 \oslash \varsigma_2 \triangleq\ & \{(l \mapsto \varsigma_1(l)) \mid l \notin \mathrm{dom}(\varsigma_2)\} \\ & \cup \{(l \mapsto \varsigma_2(l)) \mid l \in (\mathrm{dom}(\varsigma_1) \cap \mathrm{dom}(\varsigma_2))\} \\ & \cup \{(l_{\mathit{fresh}} \mapsto \varsigma_2(l)) \mid \exists l.\, l \notin \mathrm{dom}(\varsigma_1) \wedge l \in \mathrm{dom}(\varsigma_2) \wedge \\ & \qquad l_{\mathit{fresh}} \notin (\mathrm{dom}(\varsigma_1) \cup \mathrm{dom}(\varsigma_2))\}\end{aligned}$$

$$\frac{\forall l \in \mathrm{dom}(\varsigma_1).\, \varsigma_1(l) \preceq^M \varsigma_2(l)}{\varsigma_1 \preceq^{\varsigma} \varsigma_2} \qquad \frac{}{M \preceq^M M}$$

$$\frac{M_1 \preceq^M M_2 \quad M_1' \preceq^M M_2' \quad \varsigma_1 \preceq^{\varsigma} \varsigma_2 \quad \varsigma_1' \preceq^{\varsigma} \varsigma_2'}{\langle M_1, \varsigma_1 \rangle \to \langle M_1', \varsigma_1' \rangle \preceq^M \langle M_2, \varsigma_2 \rangle \to \langle M_2', \varsigma_2' \rangle}$$

Figure 3. Inductive rules for store-based typing with strong updates

	T [21]	L&W [12]	W [23]	T&J [20]	G [9]	SB
mkref	✓	✓	✓	✓	✓	✓
mkref′	–	✓	✓	✓	–	✓
effect	✓	–	–	–	✓	✓
strong	–	–	–	–	–	✓

Table 1. Comparing existing approaches and store-based typing

strong updates, $\cdot \vdash \mathit{strong}_{/\cdot} \Rightarrow^S \texttt{bool}_{/(l \mapsto \texttt{bool})}$ holds, where:

$$\mathit{strong} \triangleq \texttt{let}\ x\ \texttt{=}\ \texttt{ref}\ 1\ \texttt{in}\ x\ \texttt{:=}\ \texttt{true}$$

Table 1 summarises which expressions we are able to type check using store-based typing (SB) compared to existing approaches to imperative polymorphism in the literature.

5. Concluding Remarks

Using a novel approach to coinduction in big-step operational semantics, we have presented a method to deriving type systems that are safe by construction from big-step operational semantics, by

treating types as abstract interpretations. We propose store-based typing as an interesting avenue for further research.

5.1 Related Work

Our suggestion for dealing with divergence in Sect. 2 is closely related to the traditional approach using an $\Rightarrow^{\infty}$ predicate, which typically requires classical reasoning in proofs. Nakata and Uustalu [16] provide a constructive alternative using coinductive trace-based big-step semantics. In their approach, finite and infinite traces are distinguishable in the coinductive trace-based big-step relation. Abel and Chapman [1] encode divergence using the delay monad by wrapping computations in a coinductive type which produces a potentially infinitely-delayed value. Our encoding of divergence in a stateful way suggests that it could lend itself to implementation as a monad too.

Wright [23] provides an overview of previous approaches to Hindley-Milner polymorphism in ML-like languages. Not covered by Wright is Garrigue's more recent work [9], which uses a subtyping based approach to relax the value restriction by generalising type variables that occur only at covariant positions.

Separation logic [18] is concerned with reasoning about imperative programs and mutable data structures. Our proposed type stores reflects the runtime store. Ideas from separation logic may conceivably carry over to allow for more sophisticated store-based type analysis. Our reflection of the runtime store is also analogous to Morrisett's typed assembly language [15] which tracks the state of registers. In our system with strong updates, we expect safety to hold by the compatibility check of type stores in applications. Smith et al.'s alias types [19] instead uses linear types to track aliasing to allow for strong updates.

Ahmed [2] describes a logical relations approach to deriving type systems that are safe by construction. Like our work, she starts from an object language and derives type systems. She proves that the semantic model she uses for her derivation, based on logical relations, implies type safety. Using abstract interpretation, the appropriate definition of a Galois connection can give us a safety principle for free – an aspect explicitly mentioned by Cousot [5] in connection with the relationship between abstract interpretation and logical relations.

5.2 Future Directions

Store-based types were conceived by thinking of types as abstract interpretations, but the safety of the rules in Fig. 3 remains to be rigorously checked using the approach described in Sect. 3. From initial experiments with a Prolog prototype implementation of store-based types, we conjecture that the rules in Fig. 3 are safe.[3]

It is straightforward to restrict store-based typing to *invariant updates* such that locations in stores never change type. This would allow function types to be simplified to only contain locations that are subterms of the argument type. This may be a first step towards constructing a mapping from function monotypes with ML-style reference types into corresponding store-based type counterparts.

Abstract interpretation provides a guiding principle for constructing safe type systems. As Sect. 3 shows, it is also useful for relating type systems. An interesting line of research is to compare the expressiveness of different approaches to imperative polymorphism in the literature to our proposal.

Acknowledgments

Thanks to the referees and Neil Sculthorpe for exceptionally helpful suggestions for improving the paper. This work was supported by an EPSRC grant (EP/I032495/1) to Swansea University in connection with the *PLanCompS* project (`www.plancomps.org`).

[3] The Prolog prototype is available at: `http://www.plancomps.org/pepm2015`.

References

[1] A. Abel and J. Chapman. Normalization by evaluation in the delay monad: A case study for coinduction via copatterns and sized types. In *MSFP'14*, volume 153 of *EPTCS*, pages 51–67, 2014. doi: 10.4204/EPTCS.153.4.

[2] A. J. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[3] C. Bach Poulsen and P. D. Mosses. Deriving pretty-big-step semantics from small-step semantics. In *ESOP'14*, volume 8410 of *LNCS*, pages 270–289. Springer, 2014. doi: 10.1007/978-3-642-54833-8_15.

[4] A. Charguéraud. Pretty-big-step semantics. In *ESOP'13*, volume 7792 of *LNCS*, pages 41–60. Springer, 2013. doi: 10.1007/978-3-642-37036-6_3.

[5] P. Cousot. Types as abstract interpretations. In *POPL'97*, pages 316–331. ACM, 1997. doi: 10.1145/263699.263744.

[6] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL'79*, pages 269–282. ACM, 1979. doi: 10.1145/567752.567778.

[7] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretations. In *POPL'92*, pages 83–94. ACM, 1992. doi: 10.1145/143165.143184.

[8] L. Damas and R. Milner. Principal type-schemes for functional programs. In R. A. DeMillo, editor, *POPL'82*, pages 207–212. ACM, 1982. doi: 10.1145/582153.582176.

[9] J. Garrigue. Relaxing the value restriction. In *FLOPS'04*, volume 2998 of *LNCS*, pages 196–213. Springer, 2004. doi: 10.1007/978-3-540-24754-8_15.

[10] G. Kahn. Natural semantics. In *STACS'87*, volume 247 of *LNCS*, pages 22–39. Springer, 1987. doi: 10.1007/BFb0039592.

[11] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207:284–304, 2009. doi: 10.1016/j.ic.2007.12.004.

[12] X. Leroy and P. Weis. Polymorphic type inference and assignment. In *POPL'91*, pages 291–302. ACM, 1991. doi: 10.1145/99583.99622.

[13] B. Monsuez. Polymorphic typing by abstract interpretation. In *FSTTCS'92*, volume 652 of *LNCS*, pages 217–228. Springer, 1992. doi: 10.1007/3-540-56287-7_107.

[14] B. Monsuez. System F and abstract interpretation. In *SAS'95*, volume 983 of *LNCS*, pages 279–295. Springer, 1995. doi: 10.1007/3-540-60360-3_45.

[15] G. Morrisett. Typed assembly language. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[16] K. Nakata and T. Uustalu. Trace-based coinductive operational semantics for while. In *TPHOLs'09*, volume 5674 of *LNCS*, pages 375–390. Springer, 2009. doi: 10.1007/978-3-642-03359-9_26.

[17] B. C. Pierce. *Types and programming languages*. MIT Press, 2002.

[18] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, pages 55–74. IEEE, 2002. doi: 10.1109/LICS.2002.1029817.

[19] F. Smith, D. Walker, and G. Morrisett. Alias types. In *ESOP'00*, volume 1782 of *LNCS*, pages 366–381. Springer, 2000. doi: 10.1007/3-540-46425-5_24.

[20] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2):245–296, 1994. doi: 10.1006/inco.1994.1046.

[21] M. Tofte. Type inference for polymorphic references. *Inf. Comput.*, 89(1):1–34, Sept. 1990. doi: 10.1016/0890-5401(90)90018-D.

[22] A. K. Wright. Typing references by effect inference. In *ESOP'92*, volume 582 of *LNCS*, pages 473–491. Springer, 1992. doi: 10.1007/3-540-55253-7_28.

[23] A. K. Wright. Simple imperative polymorphism. *Lisp Symb. Comput.*, 8(4):343–355, Dec. 1995. doi: 10.1007/BF01018828.