Bidirectional Data-Flow Analyses, Type-Systematically

Maria João Frade

Departamento de Informática
Universidade do Minho
Campus de Gualtar, P-4710-057 Braga, Portugal
Email: mjf@di.uminho.pt

Ando Saabas and Tarmo Uustalu

Institute of Cybernetics
Tallinn University of Technology
Akadeemia tee 21, EE-12618 Tallinn, Estonia
Email: {ando|tarmo}@cs.ioc.ee

Abstract

We show that a wide class of bidirectional data-flow analyses and program optimizations based on them admit declarative descriptions in the form of type systems. The salient feature is a clear separation between what constitutes a valid analysis and how the strongest one can be computed (via the type checking versus principal type inference distinction). The approach also facilitates elegant relational semantic soundness definitions and proofs for analyses and optimizations, with an application to mechanical transformation of program proofs, useful in proof-carrying code. Unidirectional forward and backward analyses are covered as special cases; the technicalities in the general bidirectional case arise from more subtle notions of valid and principal types. To demonstrate the viability of the approach we consider two examples that are inherently bidirectional: type inference (seen as a data-flow problem) for a structured language where the type of a variable may change over a program's run and the analysis underlying a stack usage optimization for a stack-based low-level language.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, optimization; F.3.1 [*Logics and Meanings of Programs*]: Specifying, Verifying and Reasoning about Programs—Logics of programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational semantics, Program analysis

General Terms Languages, Theory, Verification

Keywords program analyses and optimizations, type systems, program logics, mechanical transformation of program proofs

1. Introduction

Unidirectional data-flow analyses are an important classical topic in programming theory. The theory of such analyses is well understood, there exist different styles of describing these analyses and ascribing meaning to them and their interrelationships are clear. In particular, the different styles can concentrate on the questions of what makes a valid analysis of a program or how the strongest analysis can be computed, but it is known how to relate the two aspects and there is an obvious and technically meaningful analogy to valid Hoare triples and strongest postconditions/weakest preconditions. Cascades of unidirectional analyses are enough for most analysis tasks.

Nevertheless there are also meaningful analyses that do not fit into this framework because of their inherent bidirectionality. Bidirectional analyses are considerably less well known and so is their theory. Nearly all theory on bidirectional analyses is due to Khedker and Dhamdhere [6, 7, 5], who have convincingly argued that such analyses are useful for a number of tasks, not unnatural or complicated conceptually, provided one looks at them in the right way, and not necessarily expensive to implement. However the main emphasis in this body of work has been on algorithmic descriptions that are based on transfer functions and focus on the notion of the strongest analysis of a program. By and large, these descriptions are silent about general valid analyses, which is a subtle issue in the bidirectional case, as well as semantic soundness.

In this paper, we approach bidirectional analyses with a conceptual tool that is very much oriented at a dual study of valid and strongest analyses, including semantics and soundness, in one single coherent framework, namely type systems. Type systems are a lightweight deductive means to associate properties with programs in such a way that the questions of whether a program has a given property and what the strongest property (within a given class) of the program is can be asked within the same formalism, becoming the questions of type checking versus principal type inference. We have previously argued [12, 16] that type systems are a good vehicle to describe analyses and optimizations (with type derivations as certificates of analyses of programs). This is especially true in proof-carrying code where the question of documentation and communication of analysis results is important and where type systems have an elegant application to mechanical transformation of program proofs. Similar arguments have also been put forth by Nielson and Nielson in their flow logic work [14] and Benton in his work on relational soundness [4]. Our goal here is to scale up the same technology to bidirectional analyses. We proceed from simple but archetypical examples with clear application value and arrive at several general observations.

The contribution of this paper is the following. We generalize the type-systematic account of unidirectional analyses to the bidirectional case for structured (high-level) and unstructured (low-level) languages. We formulate a schematic type system and principal type inference algorithm and show them to agree; as a side result, we show a correspondence between declarative pre/post-relations and algorithm-oriented transfer functions. Crucially, differently from unidirectional analyses, principal type inference does not mean computing the weakest pretype of a program for a given posttype, since any choice of a pretype will constrain the range of possible valid posttypes and can exclude the given one. Instead, the right generalizing notion is the weakest pre-/posttype pair for a given pre-/posttype pair. This is the greatest valid pre-/posttype

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PEPM'09, January 19–20, 2009, Savannah, Georgia, USA.
Copyright © 2009 ACM 978-1-60558-327-3/09/01…$5.00

pair pointwise smaller than the given one (which need not be valid for the given program).

Further, we show a general technique for defining soundness of analyses and optimizations based thereupon and a schematic soundness proof. This is based on similarity relations. We also demonstrate how soundness in this sense yields mechanical transformability of program proofs to accompany analyses and optimizations and argue that this is useful in proof-carrying code as a tool for the code producer.

As examples we use type inference (seen as a data-flow problem) for a structured language where a variable's type can change over a program's run but type-errors are unwanted and a stack usage optimization, namely load-pop pairs elimination, for a low-level language manipulating an operand stack. Both of these analyses are inherently bidirectional and their soundness leads to meaningful transformations of program proofs. In the first example, bidirectionality is imposed by our choice of the analysis domain (the inferred type of a variable can be either definite or unconstrained) and the notion of validity. In the second example, it is unavoidable for deeper reasons.

The load-pop pairs elimination example comes from our earlier paper [18], where we treated several stack usage optimizations. Here we elaborate this account and put it on a solid type-system-theoretic basis, discussing, in particular, type checking vs. principal type inference.

The paper is organized as follows. In Section 2, we introduce the type-systematic technique for describing bidirectional analyses on the example of type inference for a structured language. We describe this analysis declaratively and algorithmically via instances of a schematic type system and schematic principal type inference algorithm that cover a wide class of bidirectional analyses (including the standard unidirectional analyses). In Section 3, we present some basic metatheory of such descriptions: we show that pre/post-relations and transfer functions correspond and that the principal type inference algorithm is correct. In Section 4, we demonstrate the similarity-relational method to formulate soundness of analyses and give the schematic soundness proof. We also outline the application to transformation of proofs. Section 5 is to illustrate that the approach is also adaptable to unstructured languages. Here we consider load-pop pairs elimination for a stack-based language. In Section 6 we comment on some related work whereas Section 7 is to take stock and map some directions for further exploration.

2. Analyses for structured languages: type inference

In this section we introduce the type-systematic technique for describing bidirectional data-flow analyses. We do this on the example of static type inference for a language that is "dynamically" typed[1] in the sense that variable types are not declared and the type of a variable can change during a program's run. This is the simplest classical example that motivates the need for bidirectional analyses reasonably well. We present this example as an instance of the general bidirectional type-systematic framework.

The programming language we consider is WHILE. Its statements $s \in \mathbf{Stm}$, expressions $e \in \mathbf{Exp}$ are defined over a set of program variables $x \in \mathbf{Var}$ in the following way:

$$
\begin{array}{lcl}
e & ::= & x \mid \mathit{const} \mid e_0 \ \mathit{op}\ e_1 \\
s & ::= & x := e \mid \mathsf{skip} \mid s_0 ; s_1 \mid \\
 & & \mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \mid \mathsf{while}\ e\ \mathsf{do}\ s_t.
\end{array}
$$

[1] This terminology is not perfect, but it has been used, e.g., by Khedker et al. [8]. The term "dynamic" refers here to flow-sensitive typing, not to run-time type checking, i.e., variables need not have invariant types, they can have different types at different program points.

The constants const and binary operators op are drawn from a typed signature over two types int and bool (i.e., their types are $\mathit{const} : t$ and $\mathit{op} : t_0 \times t_1 \to t$ where $t, t_0, t_1 \in \{\mathrm{int}, \mathrm{bool}\}$); they are all monomorphic. A type error can occur if a guard expression is not of type bool or operands have the wrong type (e.g., at evaluating $x + y$ when x holds a value of type bool).

The type inference analysis attempts to give every variable a definite type at each program point. Intuitively, a valid analysis should have the property that if a program is started from a state where all variables satisfy the inferred pretype, then the program cannot have a runtime type error and if it finishes, the variables satisfy the inferred posttype. For example for the program $y := x; v := x + 10$, variable x should be of type int in the pretype, while y and v can have any type in the pretype. In the posttype, all variables have type int. The program $y := x; v := x \vee v; y := y{+}5$ on the other hand is ill-typed, since x is used as a bool and y as an int in the second and third assignments, but they have to be equal (and therefore have the same type) after the first assignment.

Type inference in this formulation is inherently bidirectional, meaning that information about the type of a variable at a point influences variable types both at its successor and predecessor points. A variable can only be assigned a definite type t at a program point if all reaching definitions and all reached uses agree about this. Let us look at the following program:

```
if b then
  if b' then
    x := y
  else
    x := 5
else
  w := y
```

After the first forward pass, it is known that x has type int at the end of the program, nothing is known of types of w and y. In the next backward pass, it can be found that y needs to have type int in the pretype. Using this information, running a forward pass shows that w also has type int in the posttype. While it would be possible to help type inference, e.g., by remembering equalities between variables (copy information) in addition to their types, it would still be impossible to derive the precise type by only using a forward or a backward analysis.

Type system We now state an analysis of the above-indicated flavour as a type system. The type system is given in Figures 1 and 2 where the rules in the latter are schematic for many analyses and all information specific to our particular analysis is in the former.

In our specific case, the type system features value types $\tau \in \mathbf{ValType}$ for variables, where $\mathbf{ValType} =_{\mathrm{df}} \{\mathrm{int}, \mathrm{bool}\}_{\perp}^{\top}$, $\top$ meaning "any type" and $\perp$ meaning "impossible". A state type $d \in \mathbf{StateType}$ is either a map from variables to a non-bottom value type or "impossible": $\mathbf{StateType} =_{\mathrm{df}} (\mathbf{Var} \longrightarrow \{\mathrm{int}, \mathrm{bool}\}^{\top})_{\perp}$ ($\perp$ is overloaded for variable and state types).[2] For a variable x and variable type τ, we overload notation to have $\perp[x \mapsto \tau] =_{\mathrm{df}} \perp$ and $\perp(x) =_{\mathrm{df}} \perp$. In the general case, the main category of types (that of the state types) is given by the domain D of the analysis. Subtyping follows the partial order of D.

A statement is typed with a pair of state types, the judgement syntax being $s : d \longrightarrow d'$. The intended reading is that (d, d') is a pair of agreeing pre- and poststate types for s, agreement defined via some semantic interpretation of types. In our case the interpretation is that, if the program is started in a state where the

[2] Note that the value and state types of the object language correspond to non-bottom types of the analysis type system. This particularity of our example incurs some overloading of terminology in our discussion.

$$\frac{}{\tau \leq \tau} \quad \frac{}{\bot \leq \tau} \quad \frac{}{\tau \leq \top} \quad \frac{}{\bot \leq d} \quad \frac{\forall x \in \mathbf{Var}.d(x) \leq d'(x)}{d \leq d'}$$

$$\frac{}{x : (d, d(x))} \quad \frac{d \neq \bot \quad \mathit{const} : t}{\mathit{const} : (d, t)} \quad \frac{}{\mathit{const} : (\bot, \bot)}$$

$$\frac{\mathit{op} : t_0 \times t_1 \to t \quad e_0 : (d, t_0) \quad e_1 : (d, t_1)}{e_0 \;\mathit{op}\; e_1 : (d, t)} \quad \frac{e_0 : (\bot, \bot) \quad e_1 : (\bot, \bot)}{e_0 \;\mathit{op}\; e_1 : (\bot, \bot)}$$

$$\frac{e : (d, \tau)}{x := e : d \Longrightarrow d[x \mapsto \tau]} \quad \frac{}{\mathsf{skip} : d \Longrightarrow d} \quad \frac{e : (d, \tau) \quad \tau \leq \mathrm{bool}}{e : d \Longrightarrow_{\mathsf{t}} d} \quad \frac{e : (d, \tau) \quad \tau \leq \mathrm{bool}}{e : d \Longrightarrow_{\mathsf{f}} d}$$

Figure 1. Pre/post-relations for type inference analysis

$$\frac{x := e : d \Longrightarrow d'}{x := e : d \longrightarrow d'} \text{ ass} \quad \frac{\mathsf{skip} : d \Longrightarrow d'}{\mathsf{skip} : d \longrightarrow d'} \text{ skip} \quad \frac{s_0 : d \longrightarrow d'' \quad s_1 : d'' \longrightarrow d'}{s_0; s_1 : d \longrightarrow d'} \text{ comp}$$

$$\frac{e : d \Longrightarrow_{\mathsf{t}} d_t \quad s_t : d_t \longrightarrow d' \quad e : d \Longrightarrow_{\mathsf{f}} d_f \quad s_f : d_f \longrightarrow d'}{\mathsf{if}\; e\; \mathsf{then}\; s_t\; \mathsf{else}\; s_f : d \longrightarrow d'} \text{ if} \quad \frac{e : d \Longrightarrow_{\mathsf{t}} d_t \quad s_t : d_t \longrightarrow d \quad e : d \Longrightarrow_{\mathsf{f}} d'}{\mathsf{while}\; e\; \mathsf{do}\; s_t : d \longrightarrow d'} \text{ while}$$

Figure 2. Schematic type system for general bidirectional analyses for WHILE

variables have the types specified in the pretype, then the execution, if it terminates, leads to a state where they conform with the posttype; moreover, it cannot terminate abruptly because of a type error (more on this in Section 4). Note that for simple type safety, a unidirectional analysis would suffice, but the more precise bidirectional analysis can offer additional benefits. The more a compiler knows about the possible types of a variable during the run of program, the more efficient code it can generate. If the types of all variables are predetermined for all program points, then the program can be executed taglessly: wherever some polymorphic operation (e.g., printing) is applied to some variables, it is statically known which instance of this operation is correct for this point.

The schematic typing rules for assignments and skip in Figure 2 state that they accept a pre-/posttype pair if a specific pre/post-relation $x := e : _ \Longrightarrow _$ resp. $\mathsf{skip} : _ \Longrightarrow _$ holds. The rules for if- and while-statements depend on similar relations $e : _ \Longrightarrow_{\mathsf{t}} _$ and $e : _ \Longrightarrow_{\mathsf{f}} _$ for guard expressions e. The pre/post-relations for primitive constructs given in Figure 1 are specific to our particular analysis. The relation for skip is just identity, while the relation for an assignment depends on the actual type of its right-hand-side expression e, given by the auxiliary relation $e : (_, _)$ between state and value types. In our particular case, the pre/post-relations for guard expressions for true and false branches are identical, since the analysis treats true and false branches similarly.

We have chosen to state the typing rules without a rule of subsumption. Subsumption (corresponding to laxities allowed by the analysis) is pushed up to assignments, guards and skip statements. This design gives purely syntax-directed type checking. Moreover, in the case of typical general bidirectional analyses (with identity edge flows) subsumption does not allow a statement to change its type anyhow. However our choice implies that we cannot treat skip as a trivial compound statement (with no effect in terms of data flows), but must handle it as a primitive construct.

Characteristically to data-flow analyses (but not to the standard use of type systems), all programs are typable (any program should be analyzable): it is easy to verify that any statement s can be typed at least with type $\bot \longrightarrow \bot$. However the typings of real interest are $s : d \longrightarrow d'$ where d, d' are non-bottom.

Principal type inference The type system as given in Figures 1 and 2 is purely declarative. It makes it straightforward to *check* (in a syntax-directed manner) whether a purported pre-/posttype pair for a statement s is valid. But it gives no clues for *inferring* the "best" of s. It is also an interesting question what being the "best" should mean. In the case of unidirectional analyses one typically fixes some desired posttype and asks for the weakest agreeing pretype (or vice versa). In the case of bidirectional analyses, this is not a good problem statement, as there might exist no pretype agreeing with the given posttype: any commitment about the pretype restricts the space of possible posttypes. A correct generalization would be to ask what is the greatest valid pre-/posttype pair (d, d') that is smaller than the given (not necessarily valid) pair (d_0, d'_0); we speak of computing the weakest valid pre-/posttype pair $\mathrm{wt}(s, d_0, d'_0)$.

Clearly, such a pair of types would not always exist unless the analysis had specific properties. The property needed is that D has arbitrary joins (then $D \times D$ has them too) and that the pre/post-relations for all statements in the language are closed under arbitrary joins. Then for any such relation R and any pair of given bounds (d_0, d'_0) one can identify the greatest pair (d, d') which is both in R and smaller than (d_0, d'_0). Our analysis domain has this property. But as a consequence it would, for example, not support overloaded operators: with an operator typed both $\mathrm{int} \times \mathrm{int} \to \mathrm{bool}$ and $\mathrm{bool} \times \mathrm{bool} \to \mathrm{bool}$ (such as overloaded equality), the statement $x := y \;\mathit{op}\; z$ would have no principal type for the bounding pair $(\top, \top)$. It would only have two maxima. To support such operators, a different domain must be used.

The schematic principal type inference algorithm for bidirectional type systems is given in Figure 4. It hinges on transfer functions for primitive constructions, which are specific to every particular analysis. Here the wt computation for any compound statement is a greatest fixpoint computation (unlike for unidirectional type systems, where such computations are only needed for while-loops). The bidirectionality of the algorithm is manifested in the fact that the approximations of the expected valid pre-/posttype pair recursively depend on each other as well as on the given bounding pre-/posttype pair.

The transfer functions that principal type inference relies on can be derived from the pre/post-relations of primitives, instantianting the schematic type system. The general recipe for doing this will be explained in Section 3. For our example, they are given in Figure 3. The forward and backward functions $[x := e]^{\rightarrow}$ and $[x := e]^{\leftarrow}$ for an assignment $x := e$ depend on the transfer functions $[e]^{\rightarrow}$

$$[x](d,\tau) = (d[x \mapsto d(x) \wedge \tau], d(x) \wedge \tau) \qquad [const](d,\tau) = (\textsf{if } \tau \wedge t = \bot \textsf{ then } \bot \textsf{ else } d, \tau \wedge t) \text{ for } const : t$$

$$\begin{array}{rl} [e_0 \; op \; e_1](d,\tau) = & (\textsf{if } \tau \wedge t = \bot \textsf{ then } \bot \textsf{ else } d \wedge d_0 \wedge d_1, \tau \wedge t) \text{ where} \\ & \quad (d_0, \tau_0) = [e_0](d, t_0) \\ & \quad (d_1, \tau_1) = [e_1](d, t_1) \\ & \quad \text{for } op : t_0 \times t_1 \to t \end{array}$$

$$[e]^{\leftarrow}(d,\tau) = d' \text{ where } (d', _) = [e](d,\tau) \qquad [e]^{\rightarrow} d = [e](d, \top)$$

$$\begin{array}{rcl@{\qquad}rcl} [x := e]^{\leftarrow} d' & = & [e]^{\leftarrow}(d'[x \mapsto \top], d'(x)) & [x := e]^{\rightarrow} d & = & d'[x \mapsto \tau'] \text{ where } (d', \tau') = [e]^{\rightarrow} d \\ [e]_{\mathsf{t}}^{\leftarrow} d' = [e]_{\mathsf{f}}^{\leftarrow} d' & = & [e]^{\leftarrow}(d', \mathrm{bool}) & [e]_{\mathsf{t}}^{\rightarrow} d = [e]_{\mathsf{f}}^{\rightarrow} d & = & [e]^{\rightarrow} d \\ [\mathsf{skip}]^{\leftarrow} d' & = & d' & [\mathsf{skip}]^{\rightarrow} d & = & d \end{array}$$

Figure 3. Transfer functions for type inference analysis

$$\begin{array}{rcl}
\mathrm{wt}(x := e, d_0, d_0') & =_{\mathrm{df}} & \text{greatest } (d, d') \text{ such that} \\
& & \quad d \le d_0 \wedge [x := e]^{\leftarrow} d' \\
& & \quad d' \le d_0' \wedge [x := e]^{\rightarrow} d \\
\mathrm{wt}(\mathsf{skip}, d_0, d_0') & =_{\mathrm{df}} & \text{greatest } (d, d') \text{ such that} \\
& & \quad d \le d_0 \wedge [\mathsf{skip}]^{\leftarrow} d' \\
& & \quad d' \le d_0' \wedge [\mathsf{skip}]^{\rightarrow} d \\
& = & \begin{cases} (d_0 \wedge d_0', d_0' \wedge d_0) & \text{if } [\mathsf{skip}]^{\leftarrow} = [\mathsf{skip}]^{\rightarrow} = \mathrm{id} \\ (d_0, d_0' \wedge d_0) & \text{if } [\mathsf{skip}]^{\leftarrow} = \mathrm{const}\ \top, [\mathsf{skip}]^{\rightarrow} = \mathrm{id} \\ (d_0 \wedge d_0', d_0') & \text{if } [\mathsf{skip}]^{\leftarrow} = \mathrm{id}, [\mathsf{skip}]^{\rightarrow} = \mathrm{const}\ \top \end{cases} \\
\mathrm{wt}(s_0; s_1, d_0, d_0') & =_{\mathrm{df}} & (d, d') \text{ where } (d, _, _, d') =_{\mathrm{df}} \text{greatest } (d, d'', d''', d') \text{ such that} \\
& & \quad (d, d'') \le \mathrm{wt}(s_0, d_0, d''') \\
& & \quad (d''', d') \le \mathrm{wt}(s_1, d'', d_0') \\
\mathrm{wt}(\mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f, d_0, d_0') & =_{\mathrm{df}} & (d, d') \text{ where } (d, _, _, d', _) =_{\mathrm{df}} \text{greatest } (d, d_t, d_f, d', d'') \text{ such that} \\
& & \quad d \le d_0 \wedge [e]_{\mathsf{t}}^{\leftarrow} d_t \wedge [e]_{\mathsf{f}}^{\leftarrow} d_f \\
& & \quad (d_t, d') \le \mathrm{wt}(s_t, [e]_{\mathsf{t}}^{\rightarrow} d, d_0' \wedge d'') \\
& & \quad (d_f, d'') \le \mathrm{wt}(s_f, [e]_{\mathsf{f}}^{\rightarrow} d, d_0' \wedge d') \\
\mathrm{wt}(\mathsf{while}\ e\ \mathsf{do}\ s_t, d_0, d_0') & =_{\mathrm{df}} & (d, d') \text{ where } (d, _, d') =_{\mathrm{df}} \text{greatest } (d, d_t, d') \text{ such that} \\
& & \quad (d_t, d) \le \mathrm{wt}(s_t, [e]_{\mathsf{t}}^{\rightarrow} d, d_0 \wedge [e]_{\mathsf{t}}^{\leftarrow} d_t \wedge [e]_{\mathsf{f}}^{\leftarrow} d') \\
& & \quad d' \le d_0' \wedge [e]_{\mathsf{f}}^{\rightarrow} d
\end{array}$$

Figure 4. Schematic principal type inference for WHILE

and $[e]^{\leftarrow}$ for the right-hand-side expression e (taking a state type to a pair of a state type and value type and vice versa). In the forward direction, x gets the type of the expression that is assigned to it. In the backward direction, the pretype is computed from an updated posttype where the type of x is set to be $\top$ together with the posttype of x (as the type of the expression). The reason for setting the type of x to $\top$ is that the posttype of x (the type of the new x) has no influence over the type of x during the evaluation of e (the old x). If x does not appear in the expression, its type in the pretype returned by the transfer function will remain $\top$. Otherwise, the operators in e can constrain it.

The forward transfer function $[e]^{\rightarrow}$ for an expression e takes a state type to a pair of an updated state type and a value type (a candidate type for the expression), corresponding to the idea that expression evaluation returns a value. The backward function $[e]^{\leftarrow}$ proceeds from a pair of a state type and a value type and returns an updated state type. The state type can change due to the fact that the operators have fixed types (for example for the expression $x + y$, we know that the type of the expression must be int, but also that variables x and y must have type int in the state type). If at any point a type mismatch occurs (for example, we are dealing with expression $x + y$, but x is already constrained to have type bool), it is propagated throughout and the encompassing program is ascribed the type $(\bot, \bot)$.

For the greatest fixed-points to exist, the transfer functions for the primitive constructs must be monotone (in the case of our example they are). As a consequence, all other functions whose greatest fixed-points the algorithm relies on are monotone too. The actual computation can be done by iteration, if the analysis domain has the finite descending chains property (which again holds for our example).

We should also note that unidirectional analyses, being a special case of bidirectional ones seamlessly fit in the framework. The laxities allowed by unidirectional analyses are expressed through pre/post-relations and transfer functions for assignments, guards and skip. In fact this is a good example why the typing relation for skip is not equality in the general case: in the case of unidirectional type systems, it would be $\le$ for backward analyses or $\ge$ for forward analyses. The corresponding transfer functions return constant $\top$ for the reverse direction of the analysis.

Having described the schematic type system and principal type inference algorithm on the example of type inference analysis, we now proceed to defining the mathematical relationship between the two.

3. Type checking versus principal type inference

What is required for the principal type inference algorithm to be correct with respect to the type system, i.e., to indeed compute principal types?

At the very least the principal type should always exist and the greatest fixed-points in the algorithm for finding it should exist too. Hence, the pre/post-relations ought to be closed under arbitrary joins (any subrelation of a given relation should have a join that is also in the relation) and the transfer functions must be monotone. Moreover, the transfer functions should suitably agree with the pre/post-relations. It turns out that this is enough.

Accordingly, we require that $(D, \leq)$ is a complete lattice, i.e., it has arbitrary joins $\bigvee$ (therefore also arbitrary meets). As a result $D \times D$ is also a complete lattice, with the partial order given pointwise and the join of a subrelation given by the joins of its projections: for any $R \subseteq D \times D$, we can set $\bigvee R =_{\mathrm{df}} (\bigvee R|_0, \bigvee R|_1)$ where $R|_0 =_{\mathrm{df}} \{d \mid \exists d'.\, (d, d') \in R\}$, $R|_1 =_{\mathrm{df}} \{d' \mid \exists d.\, (d, d') \in R\}$ and the operation thus defined is indeed the join of R.

Now one can switch between closed under joins relations $R \subseteq D \times D$ and pairs of monotone functions $f^{\leftarrow}, f^{\rightarrow} \in D \to D$.

We define, for a pair of monotone functions $f^{\leftarrow}$ and $f^{\rightarrow}$, a relation $\mathrm{f2R}(f^{\leftarrow}, f^{\rightarrow})$ by

$$(d, d') \in \mathrm{f2R}(f^{\leftarrow}, f^{\rightarrow}) \quad =_{\mathrm{df}} \quad d \leq f^{\leftarrow}(d') \wedge d' \leq f^{\rightarrow}(d)$$

In the opposite direction, for a joins-closed relation R, we define a pair of functions $\mathrm{R2f}^{\leftarrow}(R)$ and $\mathrm{R2f}^{\rightarrow}(R)$ by

$$\begin{array}{lll} \mathrm{R2f}^{\leftarrow}(R)(d'_0) & =_{\mathrm{df}} & \bigvee\{d \mid \exists d'.\, d' \leq d'_0 \wedge (d, d') \in R\} \\ \mathrm{R2f}^{\rightarrow}(R)(d_0) & =_{\mathrm{df}} & \bigvee\{d' \mid \exists d.\, d \leq d_0 \wedge (d, d') \in R\} \end{array}$$

We can observe the following:

THEOREM 1. *1. For any joins-closed relation R, the two functions $\mathrm{R2f}^{\leftarrow}(R)$ and $\mathrm{R2f}^{\rightarrow}(R)$ are monotone.*

2. *For any pair of monotone functions $f^{\leftarrow}$, $f^{\rightarrow}$, the relation $\mathrm{f2R}(f^{\leftarrow}, f^{\rightarrow})$ is joins-closed.*
3. *The functions $\mathrm{R2f}^{\leftarrow}, \mathrm{R2f}^{\rightarrow}$ are monotone: If $R \subseteq R'$, then $\mathrm{R2f}^{\leftarrow}(R) \leq \mathrm{R2f}^{\leftarrow}(R')$ and $\mathrm{R2f}^{\rightarrow}(R) \leq \mathrm{R2f}^{\rightarrow}(R')$.*
4. *The function $\mathrm{f2R}$ is monotone: If $f^{\leftarrow} \leq f^{\leftarrow\prime}$ and $f^{\rightarrow} \leq f^{\rightarrow\prime}$, then $\mathrm{f2R}(f^{\leftarrow}, f^{\rightarrow}) \subseteq \mathrm{f2R}(f^{\leftarrow\prime}, f^{\rightarrow\prime})$.*
5. *The functions $\langle \mathrm{R2f}^{\leftarrow}, \mathrm{R2f}^{\rightarrow} \rangle$ and $\mathrm{f2R}$ form a coreflective Galois connection: For any pair of monotone functions $f^{\leftarrow}, f^{\rightarrow}$ and joins-closed relation R, we have $\langle \mathrm{R2f}^{\leftarrow}, \mathrm{R2f}^{\rightarrow} \rangle(R) \leq (f^{\leftarrow}, f^{\rightarrow})$ if and only if $R \subseteq \mathrm{f2R}(f^{\leftarrow}, f^{\rightarrow})$.*
   Moreover, for any joins-closed relation R, $\mathrm{f2R}(\mathrm{R2f}^{\leftarrow}(R), \mathrm{R2f}^{\rightarrow}(R)) \subseteq R$.
6. *As a consequence: For any joins-closed relation R, we have $R = \mathrm{f2R}(\mathrm{R2f}^{\leftarrow}(R), \mathrm{R2f}^{\rightarrow}(R))$. For any pair of monotone functions $f^{\leftarrow}, f^{\rightarrow}$, we have $\langle \mathrm{R2f}^{\leftarrow}, \mathrm{R2f}^{\rightarrow} \rangle(\mathrm{f2R}(f^{\leftarrow}, f^{\rightarrow})) \leq (f^{\leftarrow}, f^{\rightarrow})$.*

Assuming now that for assignments, guards and the skip statement the transfer functions have been produced from their pre/post-relations with $\mathrm{R2f}^{\leftarrow}$, $\mathrm{R2f}^{\rightarrow}$, one can prove the principal type inference correct:

THEOREM 2. *$\mathrm{wt}(s, d_0, d'_0)$ is the greatest (d, d') such that $d \leq d_0$, $d' \leq d'_0$ and $s : d \longrightarrow d'$.*

The proof is by induction on the structure of s.

4. Semantic soundness and transformation of program proofs

So far we discussed the type inference analysis detached from any mathematical meaning assigned to this analysis. Now we show how the types of the analysis can be assigned an interpretation in terms of sets of states of the standard semantics of WHILE leading to a soundness result.

A state $\sigma \in \mathbf{State} =_{\mathrm{df}} \mathbf{Var} \to \mathbf{Val}$ is an assignment of values to variables where values are (tagged) integers or booleans, $\mathbf{Val} =_{\mathrm{df}} \mathbb{Z} + \mathbb{B}$. We interpret a value type τ as a subset $[\![\tau]\!]$ of $\mathbf{Val}$ by defining $[\![\bot]\!] =_{\mathrm{df}} \emptyset$, $[\![\mathrm{int}]\!] =_{\mathrm{df}} \mathbb{Z}$, $[\![\mathrm{bool}]\!] =_{\mathrm{df}} \mathbb{B}$, $[\![\top]\!] =_{\mathrm{df}} \mathbf{Val}$. This interpretation is extended to state types in the obvious pointwise way: $[\![d]\!] =_{\mathrm{df}} \{\sigma \mid \forall x \in \mathbf{Var}.\, \sigma(x) \in [\![d(x)]\!]\}$.

Let us write $\sigma \succ s \rightarrow \sigma'$ to denote that statement s run from state σ terminates in σ' and $\sigma \succ s \rightarrow\!\!\mid$ to denote that statement s run from state σ terminates abruptly. We obtain soundness in the following form: If $s : d \longrightarrow d'$ and $\sigma \in [\![d]\!]$ then (i) $\sigma \succ s \rightarrow \sigma'$ implies $\sigma' \in [\![d']\!]$ and (ii) it is not the case that $\sigma \succ s \rightarrow\!\!\mid$.

In the case of a general analysis, typically a more general approach is needed. A state type, i.e., an element d of D, is interpreted as a relation $\sim_d$ on $\mathbf{State} \times \mathbf{State}$. For type inference we define $\sigma \sim_d \sigma_*$ somewhat degenerately to mean $\sigma = \sigma_* \in [\![d]\!]$ (as a subrelation of equality). We state the soundness theorem for slightly more general type systems than we introduced thus far, with typing judgements $s : d \longrightarrow d' \hookrightarrow s_*$, expressing not only that statement s types with pre-/posttype pair d, d' but also that this typing licenses a transformation (optimization) of s into s_*. We can think of the simpler judgements $s : d \longrightarrow d'$ as abbreviations for $s : d \longrightarrow d' \hookrightarrow s$ (the transformation is identity).

THEOREM 3. *If $s : d \longrightarrow d' \hookrightarrow s_*$ and $\sigma \sim_d \sigma_*$, then (i) $\sigma \succ s \rightarrow \sigma'$ implies the existence of σ'_* such that $\sigma' \sim_{d'} \sigma'_*$ and $\sigma_* \succ s \rightarrow \sigma'_*$, and (ii) $\sigma_* \succ s_* \rightarrow \sigma'_*$ implies the existence of σ' such that $\sigma' \sim_{d'} \sigma'_*$ and $\sigma \succ s \rightarrow \sigma'$.*

Moreover, we have neither $\sigma \succ s \rightarrow\!\!\mid$ nor $\sigma_* \succ s_* \rightarrow\!\!\mid$.

The theorem is proved by structural induction on the type derivation and the only part specific to particular analyses is the base case of primitive constructs, which must be always verified specifically.

One application of relational soundness of an analysis or optimization is mechanical transformability of program proofs. This is especially useful in the case of program optimizations, as it facilitates mechanical "optimization" of a proof of a given program into one for a different program, namely the optimized form. But it is perfectly meaningful also in the case of analyses detached from any optimization, in particular, our example analysis.

A Hoare logic for a structured language where abrupt terminations are possible (because of type errors, for example) can be error-ignoring or error-free. In the error-free case, the triple $\{P\}\, s\, \{Q\}$ is intended to be derivable if and only if statement s, when started in a state satisfying P, can terminate normally only in a state satisfying Q and cannot terminate abruptly. In the error-ignoring case, no guarantees are given about impossibility of abrupt terminations. Let $P|_d$ stand for $P \wedge \bigwedge_{x \in \mathbf{Var}} x \in d(x)$, where $x \in \top =_{\mathrm{df}} \top$ and $x \in \bot =_{\mathrm{df}} \bot$. Then we get a proof transformation result: If $s : d \longrightarrow d'$ and $\{P\}\, s\, \{Q\}$ in the error-ignoring logic, then $\{P|_d\}\, s\, \{Q|_{d'}\}$ in the error-free logic. Note that the result is non-trivial: while the precondition is strengthened, the postcondition is strengthened too, and, in addition, the error-freedom guarantee is obtained.

In the case of a general analysis, one defines $P|_d$ to abbreviate $\exists w.\, w \sim_d state \wedge P[w/state]$. The general theorem is:

THEOREM 4. *If $s : d \longrightarrow d' \hookrightarrow s_*$ and $\{P\}\, s\, \{Q\}$ in the error-ignoring logic, then $\{P|_d\}\, s_*\, \{Q|_{d'}\}$ in the error-free logic.*

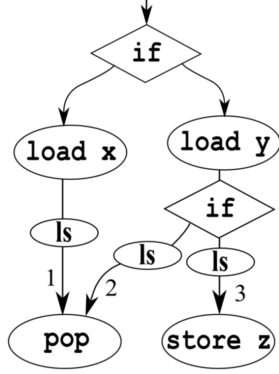

Figure 5. Example program

The theorem follows from semantic soundness of the analysis type system by the soundness and completeness of the logics. But the actual program proof transformation is obtained with a direct proof by induction on the type derivation.

5. Analyses for unstructured languages: stack usage optimizations

We now proceed to a different language without phrase structure—a stack-based language with jumps. We will show that the techniques introduced previously for structured languages apply also to flat languages where control-flow is built with jumps (essentially flowcharts). A program in such a language is essentially one big loop: instructions are fetched and executed (moving from a label to a label) until a label outside the program's support is reached and the execution is a big case distinction over the fetched instruction. It is therefore natural that a type system for an analysis is centered around big invariants which specify a condition for any label.

Before proceeding to a detailed explanation on a concrete example, let us define a simple stack-based language which we call PUSH. The building blocks of the syntax of the language are labels $\ell \in \mathbf{Label} =_{\mathrm{df}} \mathbb{N}$ (natural numbers) and instructions $instr \in \mathbf{Instr}$. We assume having a countable set of program variables $x \in \mathbf{Var}$. The instructions of the language are defined by the grammar

$$\begin{array}{rcl} instr & ::= & \mathsf{store}\ x \mid \mathsf{load}\ x \mid \mathsf{push}\ const \mid \mathsf{binop}\ op \\ & & \mid \mathsf{pop} \mid \mathsf{dup} \mid \mathsf{goto}\ \ell \mid \mathsf{gotoF}\ \ell \end{array}$$

where the constants *const* and binary operators *op* are drawn from some given signature. They are untyped, the idea being that they operate on a single set $\mathbb{W}$ of values (*words*): we do not want a possibility of errors because of wrong operand types. But a piece of code can nevertheless be unsafe as the stack can underflow (or perhaps also overflow, if there is a bound on the stack height).

A piece of code $c \in \mathbf{Code}$ is a partial finitely supported function from labels to instructions.

The example we look at is load-pop pairs elimination. Unless the optimization is restricted to load-pop pairs within basic blocks only, the underlying analysis must be bidirectional. In this general form, it was described in [18]. We repeat a large part of the description here for the sake of self-containedness.

Load-pop pairs elimination tries to find pop instructions matching up with load/push instructions and eliminate them. It makes explicit a subtlety that is present in all transformations of stack-based code that manipulate pairs of stack-height-changing instructions across basic block boundaries. This is illustrated in Figure 5, where the ls nodes denote level sequences of instructions.[3] Looking at the example, it might seem that the $\mathsf{load}\ x$ instruction can be eliminated together with pop. Closer examination reveals that this is not the case: since $\mathsf{load}\ y$ is used by $\mathsf{store}\ z$, the pop instruction cannot be removed, because then, after taking branch 2, the stack would not be balanced. This in turn means that $\mathsf{load}\ x$ cannot be removed. As can be seen from this example, a unidirectional analysis is not enough to come to such conclusion: information that a stack position is definitely needed flows backward from $\mathsf{store}\ z$ to $\mathsf{load}\ y$ along branch 3, but then the same information flows forward along path 2, and again backward along path 1. This makes the analysis inherently bidirectional, a trait common in many stack-based program analyses. Also notice that we are not really dealing with pairs, but webs of instructions in general.

In the type system, a code type $\Gamma \in \mathbf{CodeType}$ is an assignment of a state type to every label: $\mathbf{CodeType} =_{\mathrm{df}} \mathbf{Label} \to \mathbf{StateType}$. We write Γ_ℓ for $\Gamma(\ell)$. In the case of our analysis, state types are stack types plus an "impossible" type, $\mathbf{StateType} =_{\mathrm{df}} \mathbf{StackType}_\perp$ and stack types $es \in \mathbf{StackType}$ are defined by the grammar

$$\begin{array}{rcl} e & ::= & \mathrm{mnd} \mid \mathrm{opt} \\ es & ::= & [] \mid e :: es \mid * \end{array}$$

where e is a stack position type "mandatory" or "optional".

The subtyping and typing rules are given in Figure 6. A typing judgement $\Gamma \vdash c \hookrightarrow c_*$ expresses that Γ is a global invariant for c, warranting transformation (normally optimization) of c into c_*. For any label, the corresponding property holds whenever the control reaches that label, provided that the code is started so that the property corresponding to the initial label is met.

The typing rules state that, if at some label a stack position is marked "mandatory", then at all other labels of its lifetime, this particular position is also considered "mandatory". Thus the typing rules explain which optimizations are acceptable. The rule for store instructions states that the instruction always requires a "mandatory" element on the stack, thus its predecessors must definitely leave a value on top of the stack. Instructions that put elements on the stack "do not care": if an element is required, they can push a value (a mnd element on the stack in the posttype), otherwise the instruction could be omitted (an opt element on the stack in the posttype). The same holds for pop: if an element is definitely left on the stack, a pop instruction is not removed, otherwise it can be removed.

A general bidirectional analysis for PUSH would get its set of state types and the subtyping relation from a general complete lattice D; a code type then being a map $\Gamma \in \mathbf{Label} \to D$. The general type system is given in Figure 7, parameterized by joins-closed pre/post-relations for instructions. It is easy to verify that load-pop pairs analysis is an instance, and that, in particular, the pre/post-relations are joins-closed.

To obtain an algorithm for principal type inference, the relations can be turned into transfer functions following the general recipe. The monotone transfer functions are given in Figure 8. The schematic algorithm, assuming monotone transfer functions for instructions, is in Figure 9. The greatest fixed-points can again be computed by iteration, if, e.g., the domain has the finite descending chains property (in which case the iteration converges in a finite number of steps) or the transfer functions are downward ω-continuous (in which case the iteration converges at ω). Our chosen domain does not have the finite descending chains property (in-

[3] A sequence of instructions is a *level sequence*, if the net change of the stack height by these instructions is 0 and the instructions do not consume any values that were already present in the stack before executing these instructions.

$$\frac{}{\text{mnd} \le \text{opt}} \qquad \frac{}{e \le e} \qquad \frac{}{\bot \le es} \qquad \frac{}{[] \le []} \qquad \frac{e \le e' \quad es \le es'}{e :: es \le e' :: es'} \qquad \frac{}{es \le *} \qquad \frac{\forall \ell \in \mathbf{Label}.\Gamma_\ell \le \Gamma'_\ell}{\Gamma \le \Gamma'}$$

$$\frac{\Gamma_\ell = \text{mnd} :: \Gamma_{\ell+1}}{\Gamma \vdash (\ell, \mathsf{store}\ x) \hookrightarrow (\ell, \mathsf{store}\ x)}\ \text{store}$$

$$\frac{\text{mnd} :: \Gamma_\ell = \Gamma_{\ell+1}}{\Gamma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow (\ell, \mathsf{load}\ x)}\ \text{load}_1 \qquad \frac{\text{opt} :: \Gamma_\ell = \Gamma_{\ell+1}}{\Gamma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow (\ell, \mathsf{nop})}\ \text{load}_2$$

$$\frac{\text{mnd} :: \Gamma_\ell = \Gamma_{\ell+1}}{\Gamma \vdash (\ell, \mathsf{push}\ const) \hookrightarrow (\ell, \mathsf{push}\ const)}\ \text{push}_1 \qquad \frac{\text{opt} :: \Gamma_\ell = \Gamma_{\ell+1}}{\Gamma \vdash (\ell, \mathsf{push}\ const) \hookrightarrow (\ell, \mathsf{nop})}\ \text{push}_2$$

$$\frac{\Gamma_\ell = \text{mnd} :: \text{mnd} :: es \quad \text{mnd} :: es = \Gamma_{\ell+1}}{\Gamma \vdash (\ell, \mathsf{binop}\ op) \hookrightarrow (\ell, \mathsf{binop}\ op)}\ \text{binop}_1 \qquad \frac{\Gamma_\ell = \text{opt} :: \text{opt} :: es \quad \text{opt} :: es = \Gamma_{\ell+1}}{\Gamma \vdash (\ell, \mathsf{binop}\ op) \hookrightarrow (\ell, \mathsf{nop})}\ \text{binop}_2$$

$$\frac{\Gamma_\ell = \text{mnd} :: \Gamma_{\ell+1}}{\Gamma \vdash (\ell, \mathsf{pop}) \hookrightarrow (\ell, \mathsf{pop})}\ \text{pop}_1 \qquad \frac{\Gamma_\ell = \text{opt} :: \Gamma_{\ell+1}}{\Gamma \vdash (\ell, \mathsf{pop}) \hookrightarrow (\ell, \mathsf{nop})}\ \text{pop}_2$$

$$\frac{\Gamma_\ell = \text{mnd} :: es \quad \text{mnd} :: \text{mnd} :: es = \Gamma_{\ell+1}}{\Gamma \vdash (\ell, \mathsf{dup}) \hookrightarrow (\ell, \mathsf{dup})}\ \text{dup}_1 \qquad \frac{\Gamma_\ell = e_1 :: es \quad \text{opt} :: e_1 :: es = \Gamma_{\ell+1}}{\Gamma \vdash (\ell, \mathsf{dup}) \hookrightarrow (\ell, \mathsf{nop})}\ \text{dup}_2$$

$$\frac{\Gamma_\ell = \Gamma_m}{\Gamma \vdash (\ell, \mathsf{goto}\ m) \hookrightarrow (\ell, \mathsf{goto}\ m)}\ \text{goto} \qquad \frac{\Gamma_\ell = \text{mnd} :: es \quad es = \Gamma_m \quad es = \Gamma_{\ell+1}}{\Gamma \vdash (\ell, \mathsf{gotoF}\ m) \hookrightarrow (\ell, \mathsf{gotoF}\ m)}\ \text{gotoF} \qquad \frac{\Gamma_\ell = \bot \quad \bot = \Gamma_m \quad \bot = \Gamma_{\ell+1}}{\Gamma \vdash (\ell, \mathsf{gotoF}\ m) \hookrightarrow (\ell, \mathsf{gotoF}\ m)}\ \text{gotoF}$$

$$\frac{\Gamma_\ell = \bot \quad \bot = \Gamma_{\ell+1}}{\Gamma \vdash (\ell, instr) \hookrightarrow (\ell, instr)}\ \text{nonjump} \qquad \frac{\forall \ell \in \text{dom}(c).\, \Gamma \vdash (\ell, c_\ell) \hookrightarrow (\ell, c_{*\ell})}{\Gamma \vdash c \hookrightarrow c_*}\ \text{code}$$

Figure 6. Type system for load-pop pairs elimination

$$\frac{\mathsf{goto} : \Gamma_\ell \Longrightarrow \Gamma_m}{\Gamma \vdash (\ell, \mathsf{goto}\ m)}\ \text{goto} \qquad \frac{\mathsf{gotoF} : \Gamma_\ell \Longrightarrow_{\mathsf{t}} \Gamma_{\ell+1} \quad \mathsf{gotoF} : \Gamma_\ell \Longrightarrow_{\mathsf{f}} \Gamma_m}{\Gamma \vdash (\ell, \mathsf{gotoF}\ m)}\ \text{gotoF}$$

$$\frac{instr : \Gamma_\ell \Longrightarrow \Gamma_{\ell+1}}{\Gamma \vdash (\ell, instr)}\ \text{nonjump} \qquad \frac{\forall \ell \in \text{dom}(c).\, \Gamma \vdash (\ell, c_\ell)}{\Gamma \vdash c}\ \text{code}$$

Figure 7. Schematic type system for bidirectional analyses for PUSH

finite descending chains can be built from $*$), however the transfer functions are downward ω-continuous. Moreover, the algorithm still converges in a finite number of steps as soon as the bounding type for at least one label in each connected component of the code is a stack type of a specific height or $\bot$. (Also, it is possible to give the domain a finite height by bounding the stack height.)

That the algorithm really computes the principal type is expressed by the following theorem:

THEOREM 5. $\text{wt}(c, \Gamma_0)$ *is the greatest* Γ *such that* $\Gamma \le \Gamma_0$ *and* $\Gamma_0 \vdash c$.

The types can be interpreted to mean similarity relations on states of the standard semantics of the language. A state is a triple $(\ell, zs, \sigma) \in \mathbf{Label} \times \mathbf{Stack} \times \mathbf{Store}$ of a label, stack and store where a stack is a list over words and a store is an assignment of words to variables: $\mathbf{Stack} =_{\text{df}} \mathbb{W}^*$, $\mathbf{Store} =_{\text{df}} \mathbf{Var} \to \mathbb{W}$. The similarity relation is defined by the rules

$$\frac{}{[] \sim_{[]} []} \qquad \frac{zs \sim_{es} zs_*}{z :: zs \sim_{\text{mnd}::es} z :: zs_*} \qquad \frac{zs \sim_{es} zs_*}{z :: zs \sim_{\text{opt}::es} zs_*}$$

$$\frac{}{zs \sim_* []} \qquad \frac{zs \sim_{es} zs_*}{(\ell, zs, \sigma) \sim_{es} (\ell, zs_*, \sigma)}$$

The rules express that two states are related in a type, if they agree everywhere except for the optional stack positions in the first state, which must be omitted in the second. The $*$ type stands for stacks of unspecified length with all positions optional, so any stack is related to the empty stack in type $*$.

Soundness states that running the original code and its optimized form from a related pair of prestates takes them to a related pair of poststates (including equi-termination). Letting $(\ell, zs, \sigma) \succ c \rightarrow (\ell', zs', \sigma')$ to denote that code c started in state (ℓ, zs, σ) terminates in state (ℓ', zs', σ') and $(\ell, zs, \sigma) \succ c \rightarrowtail$ to denote that it terminates abruptly (because of stack underflow) (we refrain from giving the semantic evaluation rules, but they should be obvious), we can state:

THEOREM 6. *If* $\Gamma \vdash c \hookrightarrow c_*$ *and* $(\ell, zs, \sigma) \sim_{\Gamma_\ell} (\ell_*, zs_*, \sigma_*)$, *then*
(i) $(\ell, zs, \sigma) \succ c \rightarrow (\ell', zs', \sigma')$ *implies the existence of* $(\ell'_*, zs'_*, \sigma'_*)$ *such that* $(\ell', zs', \sigma') \sim_{\Gamma_{\ell'}} (\ell'_*, zs'_*, \sigma'_*)$ *and* $(\ell_*, zs_*, \sigma_*) \succ c_* \rightarrow (\ell'_*, zs'_*, \sigma'_*)$,
(ii) $(\ell_*, zs_*, \sigma_*) \succ c_* \rightarrow (\ell'_*, zs'_*, \sigma'_*)$ *implies the existence of* (ℓ', zs', σ') *such that* $(\ell', zs', \sigma') \sim_{\Gamma_{\ell'}} (\ell'_*, zs'_*, \sigma'_*)$ *and* $(\ell, zs, \sigma) \succ c \rightarrow (\ell', zs', \sigma')$.
Moreover, we have neither $(\ell, zs, \sigma) \succ c \rightarrowtail$ *nor* $(\ell_*, zs_*, \sigma_*) \succ c_* \rightarrowtail$.

Again the soundness of the analysis has a formal counterpart that can be expressed in terms of a programming logic. As mentioned earlier, this has a practical application in proof transformation, where a proof can be transformed alongside a program, guided by the same typing information.

Assume we have a program logic in the style of Bannwart and Müller [1] for reasoning about bytecode programs, with judgements $P \vdash (\ell, instr)$ for instructions and $P \vdash c$ for programs. Here, P is a map from labels to assertions, where assertions can contain the extralogical constant stk to refer to the current state of the stack. The judgement $P \vdash c$ is valid if $(zs, \sigma) \models P_\ell$ implies

$[\mathsf{store}\ x]^{\leftarrow} es$	$=$	$\mathrm{mnd} :: es$	$[\mathsf{store}\ x]^{\rightarrow}(e :: es)$	$=$	es
			$[\mathsf{store}\ x]^{\rightarrow} *$	$=$	$*$
$[\mathsf{load}\ x]^{\leftarrow}(x :: es)$	$=$	es	$[\mathsf{load}\ x]^{\rightarrow} es$	$=$	$\mathrm{opt} :: es$
$[\mathsf{load}\ x]^{\leftarrow} *$	$=$	$*$			
$[\mathsf{push}\ const]^{\leftarrow}(e :: es)$	$=$	es	$[\mathsf{push}\ const]^{\rightarrow} es$	$=$	$\mathrm{opt} :: es$
$[\mathsf{push}\ const]^{\leftarrow} *$	$=$	$*$			
$[\mathsf{binop}\ op]^{\leftarrow}(e :: es)$	$=$	$e :: e :: es$	$[\mathsf{binop}\ op]^{\rightarrow}(e :: e' :: es)$	$=$	$e \wedge e' :: es$
$[\mathsf{binop}\ op]^{\leftarrow} *$	$=$	$\mathrm{opt} :: \mathrm{opt} :: *$	$[\mathsf{binop}\ op]^{\rightarrow}(e' :: *)$	$=$	$e' :: *$
			$[\mathsf{binop}\ op]^{\rightarrow} *$	$=$	$\mathrm{opt} :: *$
$[\mathsf{pop}]^{\leftarrow} es$	$=$	$\mathrm{opt} :: es$	$[\mathsf{pop}]^{\rightarrow}(e :: es)$	$=$	es
			$[\mathsf{pop}]^{\rightarrow} *$	$=$	$*$
$[\mathsf{dup}]^{\leftarrow}(e :: e' :: es)$	$=$	$e' :: es$	$[\mathsf{dup}]^{\rightarrow}(e :: es)$	$=$	$\mathrm{opt} :: e :: *$
$[\mathsf{dup}]^{\leftarrow}(e' : *)$	$=$	$e' :: *$	$[\mathsf{dup}]^{\rightarrow} *$	$=$	$\mathrm{opt} :: \mathrm{opt} :: *$
$[\mathsf{dup}]^{\leftarrow} *$	$=$	$\mathrm{opt} :: *$			
$[\mathsf{gotoF}]_{\mathsf{t}}^{\leftarrow} es = [\mathsf{gotoF}]_{\mathsf{f}}^{\leftarrow} es$	$=$	$\mathrm{mnd} :: es$	$[\mathsf{gotoF}]_{\mathsf{t}}^{\rightarrow}(e :: es) = [\mathsf{gotoF}]_{\mathsf{f}}^{\rightarrow}(e :: es)$	$=$	es
			$[\mathsf{gotoF}]_{\mathsf{t}}^{\rightarrow} * = [\mathsf{gotoF}]_{\mathsf{f}}^{\rightarrow} *$	$=$	$*$
$[\mathsf{goto}]^{\leftarrow} es$	$=$	es	$[\mathsf{goto}]^{\rightarrow} es$	$=$	es
$[instr]^{\leftarrow} es$	$=$	$\bot$ otherwise	$[instr]^{\rightarrow} es$	$=$	$\bot$ otherwise

Figure 8. Transfer functions for load-pop pairs analysis

$$
\begin{array}{rcll}
\mathrm{wt}(c, \Gamma_0) & =_{\mathrm{df}} & \text{largest } \Gamma \text{ such that } \Gamma_\ell & \leq \Gamma_{0\ell} \\
& & & \wedge\ \bigwedge\{[instr]^{\leftarrow}\Gamma_{\ell+1} \mid c_\ell = instr\} \\
& & & \wedge\ \bigwedge\{[\mathsf{goto}]^{\leftarrow}\Gamma_m \mid c_\ell = \mathsf{goto}\ m\} \\
& & & \wedge\ \bigwedge\{[\mathsf{gotoF}]_{\mathsf{t}}^{\leftarrow}\Gamma_{\ell+1} \wedge [\mathsf{gotoF}]_{\mathsf{f}}^{\leftarrow}\Gamma_m \mid c_\ell = \mathsf{gotoF}\ m\} \\
& & & \wedge\ \bigwedge\{[instr]^{\rightarrow}\Gamma_{\ell-1} \mid c_{\ell-1} = instr\} \\
& & & \wedge\ \bigwedge\{[\mathsf{goto}]^{\rightarrow}\Gamma_m \mid c_m = \mathsf{goto}\ \ell\} \\
& & & \wedge\ \bigwedge\{[\mathsf{gotoF}]_{\mathsf{t}}^{\rightarrow}\Gamma_{\ell-1} \mid c_{\ell-1} = \mathsf{gotoF}\ m\} \\
& & & \wedge\ \bigwedge\{[\mathsf{gotoF}]_{\mathsf{f}}^{\rightarrow}\Gamma_m \mid c_m = \mathsf{gotoF}\ \ell\}
\end{array}
$$

Figure 9. Schematic principal type inference for PUSH

that (i) $(\ell, zs, \sigma) \succ c \rightarrow (\ell', zs', \sigma')$ implies $(zs', \sigma') \models P_{\ell'}$ and (ii) in the case of an error-free logic, also that $(\ell, zs, \sigma) \succ c \twoheadrightarrow$ cannot be.

It is then easy to show that if $\Gamma \vdash c \hookrightarrow c_*$ then any proof for $P \vdash c$ can be transformed into a corresponding proof for $P|_\Gamma \vdash c_*$, where $(P|_\Gamma)_\ell =_{\mathrm{df}} \exists w.\ w \sim_{\Gamma_\ell} stk \wedge P_\ell[w/stk]$.

Informally, each $(P|_\Gamma)_\ell$ is obtained from P_ℓ by quantifying out stack positions which are opt, i.e., stack values which are absent in the optimized program. Of course this changes the height of the stack, so any stack position below the removed one is shifted up.

For example, if we have an assertion $P|_\ell =_{\mathrm{df}} \exists v.\ st = v :: 6 :: [] \wedge 2 * v = x$ and type $\Gamma_\ell = \mathrm{opt} :: \mathrm{mnd} :: []$, the assertion $(P|_\Gamma)_\ell$ becomes $\exists v. st = 6 :: [] \wedge 2 * v = x$.

We obtain the following proof transformation theorem.

THEOREM 7. *If $\Gamma \vdash c \hookrightarrow c_*$ and $P \vdash c$ in the error-ignoring logic, then $P|_\Gamma \vdash c_*$ in the error-free logic.*

6. Related Work

We proceeded from our own work on type systems for analyses and optimizations [12, 16, 9, 18, 17], with applications, in particular, to program proof transformation, but similar techniques appear in a number of works where semantics is a concern. Most relevantly for us here, the distinction between declarative and algorithmic is prevalent in the flow logic work of Nielson and Nielson [14]. In this terminology, our exposition of the type inference analysis is in the "compositional, succinct format" ('compositional' referring to working on the phrase structure, 'succinct' to not annotating inner points of a phrase) while the treatment of the load-pop pairs analysis is in the "abstract" format ('abstract' referring to working on a flow chart representation). Semantic soundness based on similarity relations has a central role for Benton [4]. Systematic optimization soundness proofs are the central concern in the work of Lerner et al. on the Rhodium DSL for describing program analyses and optimizations [13]. Transformation of program proofs has also been considered by Barthe et al. [2, 3], but their approach cannot handle general similarity relations. In our terms, it is confined to similarity relations that are subrelations of equality; in proof transformation, assertions are accordingly only strengthened.

Static type inference for a "dynamically" typed imperative language is a classical problem. In particular, it has been understood as a bidirectional data-flow problem at least since Tenenbaum [19], Jones and Muchnick [10] and Kaplan and Ullman [11]. There exist very fine bidirectional analyses, e.g., a relatively recent one by Khedker et al. [8], but the domain of the inferrable types and its interpretation varies a lot. Also, far from always is it clear what the intended notion of validity of an analysis is intended to be. We consider a rather basic analysis with a very simple domain, but it is nevertheless instructive and (more importantly) sound wrt. a very useful semantics: a variable acquiring a type at some point means that all reaching definitions and all future uses before future redefinitions agree with this type, guaranteeing safety and enabling tagless execution.

Inferring stack types is an integral element in Java bytecode verification. A load-pop pairs removal analysis has been proposed and proved correct by Van Drunen et al. [20], but only for straight-line programs (no jumps). We have treated the general case as a bidirectional analysis and optimization [18], covering also proof transformation [15].

7. Conclusions and Future Work

Our goal with this paper was to produce an account of bidirectional data-flow analyses that enables a clear distinction between acceptable analyses and the strongest analysis of a program. We chose to try to base such an account on type-systematic descriptions where this distinction is inherent. We deem that this attempt was successful: type systems provide indeed a useful way to look at bidirectional analyses. Here is why.

Bidirectional analyses have been defined and assessed mostly algorithmically in ways concentrating on the strongest analysis algorithm of a program and tending to leave it vague what general valid analyses would be. This breaks the natural modular organization of the metatheory where the soundness statement of an analysis pertains to any valid analysis of a program and the soundness of the strongest analysis is only one (trivial) consequence.

In contrast, in the type-systematic account, the notion of a general valid analysis is central and primary. The strongest analysis becomes a derived notion and a nice correspondence between pre/post-relations (defining general analyses) and transfer functions (instrumental in algorithms for computing the strongest analyses) arises. From the point of view of trusting analyses computed by another party (useful in proof-carrying code applications), it is clearly beneficial to be able to determine whether a purported analysis result is valid without having to know how it was computed or to recompute it.

The future work will address a number of issues that we refrained from treating here. We did not show that, for high-level programs, the structured (declarative resp. algorithmic) definitions of a valid analysis and the strongest analysis agree with the corresponding flat definitions on the control-flow graph. We did not show that analysis domain elements can normally be understood as properties of computation traces/trees (stretching, in the case of bidirectional analyses, to both the past and the future), leading to results of soundness and completeness wrt. accordingly abstracted semantics (completeness holds only for distributive transfer functions). We did comment on the relationship of this semantics to similarity relations. Our comments on mechanical program transformation were only tangential.

In terms of the reach of approach, we made some deliberate simplifications in this paper. In particular, we chose to hide edge flows into node flows (making it necessary to see skip statements and goto instructions as nodes rather than "nothing" in terms of control flow). For complex bidirectional analyses, this is not an option. An alternative and more scalable approach is to support edge flows directly by an explicit consequence (subsumption) rule in the case of a structured language and by separate node exit and entry points in the case of an unstructured language.

Acknowledgments

We are thankful for our three anonymous PEPM 2009 referees for their useful remarks.

This work was supported by the Portuguese Foundation for Science and Technology under grant No. FCT/PTDC/EIA/65862/2006 RESCUE, the Estonian Science Foundation under grant No. 6940 and the EU FP6 IST integrated project No. 15905 MOBIUS.

References

[1] F. Bannwart, P. Müller. A program logic for bytecode. In *Proc. of 1st Wksh. on Bytecode Semantics, Verification, Analysis and Transformation, Bytecode 2005*, v. 141(1) of *Electron. Notes in Theor. Comput. Sci.*, pp. 255–273, Elsevier, 2005

[2] G. Barthe, B. Grégoire, C. Kunz, T. Rezk. Certificate translation for optimizing compilers. In K. Yi, ed., *Proc. of 13th Int. Symp. on Static Analysis, SAS 2006*, v. 4134 of *Lect. Notes in Comput. Sci.*, pp. 301–317, Springer, 2006.

[3] G. Barthe, C. Kunz. Certificate translation in abstract interpretation. In S. Drossopoulou, ed., *Proc. of 17th Europ. Symp. on Programming, ESOP 2008*, v. 4960 of *Lect. Notes in Comput. Sci.*, pp. 368–382, Springer, 2008.

[4] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proc. of 31st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 2004*, pp. 14–25, ACM Press, 2004.

[5] U. P. Khedker. Data flow analysis. In Y. N. Srikant, P. Shankar, eds., *The Compiler Design Handbook: Optimization and Machine Code Generation*, pp. 1–59. CRC Press, 2002.

[6] U. P. Khedker, D. M. Dhamdhere. A generalized theory of bit vector data flow analysis. *ACM Trans. on Program. Lang. and Syst.*, 16(5):1472–1511, 1994.

[7] U. P. Khedker, D. M. Dhamdhere. Bidirectional data flow analysis: myths and reality. *ACM SIGPLAN Notices*, 34(6):47–57, 1999.

[8] U. P. Khedker, D. M. Dhamdhere, A. Mycroft. Bidirectional data flow analysis for type inferencing. *Comput. Lang., Syst. and Struct.*, 29(1–2):15–44, 2003.

[9] M. J. Frade, A. Saabas, T. Uustalu. Foundational certification of data-flow analyses. In *Proc. of 1st IEEE and IFIP Int. Symp on Theor. Aspects of Software Engineering, TASE 2007*, pp. 107–116, IEEE CS Press, 2007.

[10] N. D. Jones, S. S. Muchnick. Binding time optimization in programming languages: some thoughts toward the design of an ideal language. In *Proc. of 3rd ACM Symp. on Principles of Programming Languages, POPL 1976*, pp. 77–94, ACM Press, 1976.

[11] M. A. Kaplan, J. D. Ullman. A scheme for the automatic inference of variable types. *J. of ACM*, 27(1):128–145, 1980.

[12] P. Laud, T. Uustalu, V. Vene. Type systems equivalent to data-flow analyses for imperative languages. *Theor. Comput. Sci.*, 364(3):292–310, 2006.

[13] S. Lerner, T. Millstein, E. Rice, C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proc. of 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 2005*, pp. 364–377, ACM Press, 2005.

[14] H. R. Nielson, F. Nielson. Flow logic: a multi-paradigmatic approach to static analysis. In T. Æ. Mogensen, D. A. Schmidt, I. H. Sudborough, eds., *The Essence of Computation, Complexity, Analysis, Transformation*, v. 2566 of *Lect. Notes in Comput. Sci.*, pp. 223–244, Springer-Verlag, 2002.

[15] A. Saabas. *Logics for low-level code and proof-preserving program transformations* (PhD thesis), *Thesis on Informatics and System Engineering* C143. Tallinn Univ. of Techn., 2008.

[16] A. Saabas, T. Uustalu. Program and proof optimizations with type systems. *J. of Logic and Algebraic Program.*, 77(1–2):131–154, 2008.

[17] A. Saabas, T. Uustalu. Proof optimization for partial redundancy elimination. In *Proc. of 2008 ACM SIGPLAN Wksh. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 2008*, pp. 91–101. ACM Press, 2008

[18] A. Saabas, T. Uustalu. Type systems for optimizing stack-based code. In M. Huisman, F. Spoto, eds., *Proc. of 2nd Int. Wksh. on Bytecode Semantics, Verification, Analysis and Transformation, Bytecode 2007*, v. 190(1) of *Electron. Notes in Theor. Comput. Sci.*, pp. 103–119, Elsevier, 2007. (The treatment of $*$ in the accounts of dead stores and load-pop pairs elimination is garbled in the published version.)

[19] A. Tenenbaum. Type determination for very high level languages. Report NSO-3, Courant Inst. of Math. Sci., New York Univ., New York, 1974.

[20] T. Van Drunen, A. L. Hosking, J. Palsberg. Reducing loads and stores in stack architectures, manuscript, 2000.