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 pointwise smaller than the one given (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 \(s \in \text{Stm}\), expressions \(e \in \text{Exp}\) are defined over a set of program variables \(x \in \text{Var}\) in the following way:

\[
e := x \mid \text{const} \mid e_0 \; \text{op} \; e_1
\]

\[
s := x := e \mid \text{skip} \mid s_0; s_1
\]

\[
\text{if } e \text{ then } s_1 \text{ else } s_f \mid \text{while } e \text{ do } s_i
\]

The constants \(\text{const}\) and binary operators \(\text{op}\) are drawn from a typed signature over two types int and bool (i.e., their types are \(\text{const} : \top\) and \(\text{op} : t_0 \times t_1 \rightarrow t \) where \(t, t_0, t_1 \in \{\text{int, bool}\}\); 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 if and only if, 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 \(y := x; v := x \land v; y := y + 5\) 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:

\[
\begin{align*}
\text{if } b \text{ then }& \\
\text{if } b' \text{ then }& \quad x := y \\
\text{else }& \quad x := 5 \\
\text{else }& \quad w := y
\end{align*}
\]

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 flavor 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 \(\tau \in \text{ValType}\) for variables, where \(\text{ValType} =_{\text{df}} \{\text{int, bool}\}^\top\). \top meaning “any type” and \(\bot\) meaning “impossible”. A state type \(d \in \text{StateType}\) is either a map from variables to a non-bottom value type or “impossible”: \(\text{StateType} =_{\text{df}} (\text{Var} \rightarrow (\text{int, bool})^\top)\). \bot is overloaded for variable and state types.\(^2\)

For a variable \(x\) and variable type \(\tau\), we overload notation to have \(\bot(x \rightarrow \tau) =_{\text{df}} \bot\) and \(\bot(x) =_{\text{df}} \bot\). 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 \(s : d \rightarrow d'\). 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

\(^1\)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 runtime type checking, i.e., variables need not have invariant types, they can have different types at different program points.

\(^2\)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.
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 when viewing Figure 2. It can be derived from the pre/post-relations of primitives, instantiating the schematic type system for general bidirectional analyses for WHILE

```
\[
\begin{array}{l}
\forall x \in \text{Var}. d(x) \leq d'(x) \\
\forall \tau \leq \tau' \quad \bot \leq \tau' \quad \tau \leq \top \quad \bot \leq d \quad d \leq d' \\
\frac{x : (d, d(x))}{\text{const} : t} \quad \frac{d \neq \bot \text{ const}}{\text{const} : (d, t)} \\
\frac{e_0 : (d, t_0) \quad e_1 : (d, t_1)}{e_0 \text{ op } e_1 : (d, t)} \\
\frac{e_0 : (\bot, \bot) \quad e_1 : (\bot, \bot)}{e_0 \text{ op } e_1 : (\bot, \bot)} \\
\frac{e : (d, \tau)}{x := e : d \Rightarrow d[x \mapsto \tau]} \\
\frac{\text{skip} : d \Rightarrow d}{e : (d, \tau) \quad \tau \leq \text{bool}} \\
\frac{e : (d, \tau) \quad \tau \leq \text{bool}}{e : d \Rightarrow \bot} \\
\end{array}
\]

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

```
\[
\begin{array}{l}
x := e : d \Rightarrow d' \\
x := e : d \Rightarrow d' \\
\frac{\text{skip} : d \Rightarrow d'}{\text{skip} : d \Rightarrow d'} \\
\frac{e : d \Rightarrow \tau \quad d_t \quad s_t : d_t \Rightarrow d'}{e : d \Rightarrow \tau} \\
\frac{s_f : d \Rightarrow d'}{\text{while} (e \Rightarrow \tau) \quad d \Rightarrow \tau} \\
\end{array}
\]

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

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 $\mathcal{R}$ and any pair of given bounds $(d_0, d')$ one can identify the greatest pair $(d, d')$ which is both in $\mathcal{R}$ and smaller than $(d_0, d')$. Our analysis domain has this property. But as a consequence it would, for example, not support overloaded operators: with an operator typed both $\text{int} \times \text{int} \Rightarrow \text{bool}$ and $\text{bool} \times \text{bool} \Rightarrow \text{bool}$ (such as overloaded equality), the statement $x := y \text{ op } z$ 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/post-type pair recursively depend on each other as well as on the given bounding pre/post-type pair.

The transfer functions that principal type inference relies on can be derived from the pre/post-relations of primitives, instantiating 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 $[x := e]^-\tau$ and $[x := e]^-\tau$ for an assignment $x := e$ depend on the transfer functions $[e]^-\tau$.
expression evaluation returns a value. The backward function \( [e]^{-} \) corresponds to the idea that the state type to a pair of an updated state type and a value type (a candidate type for the expression), \( \tau \) where

\[
[x](d, \tau) = (d[x \leftarrow d(x) \land \tau], d(x) \land \tau) \quad \text{[const]}(d, \tau) = (\text{if } \tau \land t = \bot \text{ then } \bot \text{ else } d, \tau \land t) \text{ for } \text{const} : t
\]

\[e_0 \text{ op } e_1(d, \tau) = (\text{if } \tau \land t = \bot \text{ then } \bot \text{ else } d_0 \land d_1, \tau \land t)\]

for \( \text{op} : t_0 \times t_1 \rightarrow t \)

\[ [e]^{-} = \text{d} \text{ where } (\text{d}', \rho) = [e](d, \tau) \quad [e]^{-}d = [e](d, \top) \]

\[ [x := e]^{-}d' = [e]^{-}d'(x \leftarrow \top), d'(x)] \quad [x := e]^{-}d = d'[x \leftarrow \tau'] \text{ where } (d', \tau') = [e]^{-}d \]

\[ [\text{skip}]^{-}d' = [e]^{-}(d', \text{bool}) \quad [\text{skip}]^{-}d = d \]

<table>
<thead>
<tr>
<th>( \text{wt} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := e, d_0, d_0' )</td>
</tr>
<tr>
<td>( =_{\text{df}} )</td>
</tr>
<tr>
<td>greatest ( (d, d') ) such that</td>
</tr>
<tr>
<td>( d \leq d_0 \land [x := e]^{-}d' )</td>
</tr>
<tr>
<td>( d' \leq d_0' \land [x := e]^{-}d )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \text{wt} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{skip}, d_0, d_0' )</td>
</tr>
<tr>
<td>( =_{\text{df}} )</td>
</tr>
<tr>
<td>greatest ( (d, d') ) such that</td>
</tr>
<tr>
<td>( d \leq d_0 \land [\text{skip}]^{-}d' )</td>
</tr>
<tr>
<td>( d' \leq d_0' \land [\text{skip}]^{-}d )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \text{wt} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_0; s_1, d_0, d_0' )</td>
</tr>
<tr>
<td>( =_{\text{df}} )</td>
</tr>
<tr>
<td>( (d, d') ) where ( (d, \ldots, d') =_{\text{df}} ) greatest ( (d, d', d'', d') ) such that</td>
</tr>
<tr>
<td>( (d', d'') \leq \text{wt}(s_0, d_0, d'') )</td>
</tr>
<tr>
<td>( (d'', d') \leq \text{wt}(s_1, d', d_0') )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \text{wt} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{if } e \text{ then } s_1 \text{ else } s_f, d_0, d_0' )</td>
</tr>
<tr>
<td>( =_{\text{df}} )</td>
</tr>
<tr>
<td>( (d, d') ) where ( (d, \ldots, d', \ldots) =_{\text{df}} ) greatest ( (d, d, d', d') ) such that</td>
</tr>
<tr>
<td>( d \leq d_0 \land [e]^{-}d \land [e]^{-}d_f \land [e]^{-}d )</td>
</tr>
<tr>
<td>( (d, d') \leq \text{wt}(s_1, [e]^{-}d, d_0 \land d'') )</td>
</tr>
<tr>
<td>( (d', d') \leq \text{wt}(s_f, [e]^{-}d, d_0 \land d'') )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \text{wt} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{while } e \text{ do } s_t, d_0, d_0' )</td>
</tr>
<tr>
<td>( =_{\text{df}} )</td>
</tr>
<tr>
<td>( (d, d') ) where ( (d, \ldots, d', \ldots) =_{\text{df}} ) greatest ( (d, d, d', d') ) such that</td>
</tr>
<tr>
<td>( (d, d') \leq \text{wt}(s_1, [e]^{-}d, d_0 \land [e]^{-}d_t \land [e]^{-}d'') )</td>
</tr>
<tr>
<td>( (d', d') \leq d_0' \land [e]^{-}d' )</td>
</tr>
</tbody>
</table>

and \( [e]^{-} \) 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 \( [e]^{-} \) 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), \( \tau \) corresponding to the idea that expression evaluation returns a value. The backward function \( [e]^{-} \) 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 \( (\ldots, \bot) \).

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 \( \leq \) for backward analyses or \( \geq \) 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 \(\lor\) (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 \(\forall R = \{ (d, d') \mid (d, d') \in R \}\), \(R_1 = \{ d' \mid \exists d, (d, d') \in R \}\) 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 \(f^{-}, f^{+}\), where the transfer functions have been produced from their pre/post-relations 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 \(\lor\) (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 \(\forall R = \{ (d, d') \mid (d, d') \in R \}\), \(R_1 = \{ d' \mid \exists d, (d, d') \in R \}\) 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 \(f^{-}, f^{+}\), where the transfer functions have been produced from their pre/post-relations 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 \(\lor\) (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 \(\forall R = \{ (d, d') \mid (d, d') \in R \}\), \(R_1 = \{ d' \mid \exists d, (d, d') \in R \}\) 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 \(f^{-}, f^{+}\), where the transfer functions have been produced from their pre/post-relations 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 \(\lor\) (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 \(\forall R = \{ (d, d') \mid (d, d') \in R \}\), \(R_1 = \{ d' \mid \exists d, (d, d') \in R \}\) 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 \(f^{-}, f^{+}\), where the transfer functions have been produced from their pre/post-relations 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 \(\lor\) (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 \(\forall R = \{ (d, d') \mid (d, d') \in R \}\), \(R_1 = \{ d' \mid \exists d, (d, d') \in R \}\) 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 \(f^{-}, f^{+}\), where the transfer functions have been produced from their pre/post-relations 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 \(\lor\) (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 \(\forall R = \{ (d, d') \mid (d, d') \in R \}\), \(R_1 = \{ d' \mid \exists d, (d, d') \in R \}\) 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 \(f^{-}, f^{+}\), where the transfer functions have been produced from their pre/post-relations 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 \(\lor\) (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 \(\forall R = \{ (d, d') \mid (d, d') \in R \}\), \(R_1 = \{ d' \mid \exists d, (d, d') \in R \}\) 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 \(f^{-}, f^{+}\), where the transfer functions have been produced from their pre/post-relations 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 \(\lor\) (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 \(\forall R = \{ (d, d') \mid (d, d') \in R \}\), \(R_1 = \{ d' \mid \exists d, (d, d') \in R \}\) 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 \(f^{-}, f^{+}\), where the transfer functions have been produced from their pre/post-relations 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 \(\lor\) (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 \(\forall R = \{ (d, d') \mid (d, d') \in R \}\), \(R_1 = \{ d' \mid \exists d, (d, d') \in R \}\) and the operation thus defined is indeed the join of \(R\).
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 $\ell \in \text{Label}$, instructions $\text{instr} \in \text{Instr}$, and variables $x \in \text{Var}$. The instructions of the language are defined by the grammar

$$\text{instr} ::= \text{store } x | \text{load } x | \text{push } \text{const} | \text{binop } \text{op}$$

where the constants $\text{const}$ and binary operators $\text{op}$ are drawn from some given signature. They are untyped, the idea being that they operate on a single set $\mathcal{W}$ 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 $c \in \text{Code}$ 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. Looking at the example, it might seem that the load $x$ instruction can be eliminated together with pop. Closer examination reveals that this is not the case: since load $y$ is used by 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 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 store $z$ to 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 $\Gamma \in \text{CodeType}$ is an assignment of a state type to every label: $\text{CodeType} = \text{af Label} \rightarrow \text{StateType}$. We write $\Gamma_\ell$ for $\Gamma(\ell)$. In the case of our analysis, state types are stack types plus an “impossible” type. $\text{StateType} = \text{af StackType}$ and stack types $\text{es} \in \text{StackType}$ are defined by the grammar

$$e ::= \text{mnd} \mid \text{opt}$$

$$\text{es} ::= [] \mid e :: \text{es} \ast$$

where $e$ is a stack position type “mandatory” or “optional”.

The subtyping and typing rules are given in Figure 6. A typing judgement $\Gamma \vdash c \rightarrow c_*$ expresses that $\Gamma$ 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 complete lattice $D$; a code type then being a map $\Gamma \in \text{Label} \rightarrow D$. 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 $\omega$-continuous (in which case the iteration converges at $\omega$). 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.
finite descending chains can be built from \(*\), however the transfer functions are downward \(\omega\)-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 \(\Gamma\) such that \(\Gamma \leq \Gamma_0\) 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 \((\ell, zs, \sigma)\) \(\in\) \(\text{Label} \times \text{Stack} \times \text{Store}\) of a label, stack and store where a stack is a list over words and a store is an assignment of words to variables: \(\text{Stack} =_{df} \mathbb{W}^*, \text{Store} =_{df} \text{Var} \rightarrow \mathbb{W}\). The similarity relation is defined by the rules:

\[
\begin{align*}
\{(\ell, zs, \sigma) \sim_{cs} (\ell', zs', \sigma')\} & \quad 2s \sim_{cs} 2s' & \quad 2s \sim_{cs} 2s' \quad 2s \sim_{cs} 2s' & \quad 2s \sim_{cs} 2s' \\
\{(\ell, zs, \sigma) \sim_{cs} (\ell, zs', \sigma')\} & \quad (\ell, zs, \sigma) \sim_{cs} (\ell, zs, \sigma')
\end{align*}
\]

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 \((\ell, zs, \sigma) \sim_{c} (\ell', zs', \sigma')\) to denote that code \(c\) started in state \((\ell, zs, \sigma)\) terminates in state \((\ell', zs', \sigma')\) and \((\ell, zs, \sigma) \sim_{c} \tau\) 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 \(\Gamma_0 \vdash c \leftarrow c_0\) and \((\ell, zs, \sigma) \sim_{\Gamma_0} (\ell_0, zs_0, \sigma_0)\),

(i) \((\ell, zs, \sigma) \sim_{c} (\ell', zs', \sigma')\) implies the existence of \((\ell', zs', \sigma') \sim_{\Gamma_0} (\ell'_0, zs'_0, \sigma'_0)\) and \((\ell, zs, \sigma) \sim_{c} \tau\) to \((\ell', zs', \sigma')\),

(ii) \((\ell, zs, \sigma) \sim_{c} (\ell', zs', \sigma')\) implies the existence of \((\ell', zs', \sigma') \sim_{\Gamma_0} (\ell'_0, zs'_0, \sigma'_0)\) and \((\ell, zs, \sigma) \sim_{c} \tau\) to \((\ell', zs', \sigma')\).

Moreover, we have neither \((\ell, zs, \sigma) \sim_{c} \tau\) to \((\ell, zs, \sigma)\).

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 Bannwert 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 \(\text{stk}\) to refer to the current state of the stack. The judgement \(P \vdash c\) is valid if \((zs, \sigma) \models P\) implies...
that (i) \((\ell, zs, \sigma) \Rightarrow c \Rightarrow (\ell', zs', \sigma')\) implies \((zs', \sigma') \models P_{\ell'}\) and (ii) in the case of an error-free logic, also that \((\ell, zs, \sigma) \Rightarrow c \Rightarrow \) cannot be.

It is then easy to show that if \(\Gamma \vdash c \leftarrow c_{\ell}\), then any proof for \(P \vdash c\) can be transformed into a corresponding proof for \(P_{\ell} \vdash c_{\ell}\), where \(P_{\ell} \models_{dt} \exists \Gamma, st \vdash v = 6\) and \(P_{\ell} \models_{dt} \Gamma_{s_{\ell}} \vdash \Gamma_{c_{\ell}} \models_{dt} c_{\ell} = instr\).

In formally, each \(P_{\ell} \models_{dt} \) is obtained from \(P_{\ell} \) 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 \(P_{\ell} \models_{dt} \exists \Gamma, st \vdash v = 6\) and \(P_{\ell} \models_{dt} \Gamma_{s_{\ell}} \vdash \Gamma_{c_{\ell}} \models_{dt} c_{\ell} = instr\)

We obtain the following proof transformation theorem.

**Theorem 7.** If \(\Gamma \vdash c \leftarrow c_{\ell}\) and \(P \vdash c\) in the error-ignoring logic, then \(P_{\ell} \vdash c_{\ell}\) 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 subreductions 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 inferable 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 re-definitions agree with this type, guaranteeing safety and enabling tagless execution.

Infering stack types is an integral element in Java bytecode verification. A load-pop pairs removal analysis has been proposed and proved correct by Van Druyen et al. [20], but only for straightforward 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 (in instrumental 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 edge flows directly by an explicit consequence (subsumption) rule.

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.

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.

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


